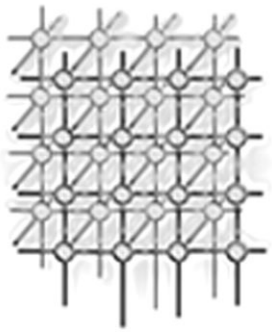
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
*Concurrency Computat.: Pract. Exper.* 2003; **15**:155–179 (DOI: 10.1002/cpe.714)

# Improving the official specification of Java bytecode verification



Alessandro Coglio*,†

*Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA 94304, U.S.A.*

**SUMMARY**

**Bytecode verification is the main mechanism to ensure type safety in the Java Virtual Machine. Inadequacies in its official specification may lead to incorrect implementations where security can be broken and/or certain legal programs are rejected. This paper provides a comprehensive analysis of the specification, along with concrete suggestions for improvement. Copyright © 2003 John Wiley & Sons, Ltd.**

KEY WORDS: Java; bytecode verification; specification

## 1. INTRODUCTION

Java programs [1,2] are normally compiled to class files, which are executed by the Java Virtual Machine (JVM) [3]. A class file contains substantially the same information as the Java class or interface from which it is generated, except that each method's code is replaced by a sequence of platform-independent, assembly-like instructions called 'bytecodes'‡. Bytecodes are typically executed by direct interpretation or just-in-time compilation to native machine code.

Java is designed to be type-safe, i.e. the type of a value determines the operations allowed on the value. For instance, to forge object references from integers or to access arrays outside their bounds is not allowed. Besides supporting a better programming discipline and making programs more robust, type safety is the basis of Java security [4].

Java compilers perform most of the needed type safety checks; only those properties that cannot be statically checked in general (e.g. that arrays are accessed within their bounds) are deferred to run time. Nonetheless, the JVM has typically no guarantees that incoming class files (e.g. Web applets from

*Correspondence to: Alessandro Coglio, Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA 94304, U.S.A.
†E-mail: coglio@kestrel.edu
‡The term 'bytecode' is often abused so as to denote the language of bytecodes, including the ancillary information in class files. This paper also conforms to this abuse.


*Received 8 January 2002*
*Revised 17 August 2002*

remote Internet sites) have been generated by trustworthy compilers. The JVM must independently ensure type safety, without relying on compilers.

If a Java program is successfully compiled, the JVM must be able to re-establish the type safety properties established by the compiler. It would be disappointing if the JVM refused to run the generated code due to the 'violation' of a property that the compiler has assessed as satisfied.

Bytecode verification is the main mechanism to ensure type safety in the JVM. Prior to execution, incoming code is screened by the bytecode verifier in order to establish that certain type safety properties will be satisfied when the code is run. This relieves the JVM from checking such properties at run time, dramatically improving performance.

Since Java security is based on type safety, a correct implementation of bytecode verification is of paramount importance to the security of an implementation of the JVM. Holes in bytecode verification constitute potential points of attack exploitable by malicious programs to overthrow the other security mechanisms of the JVM [5]. A correct implementation of bytecode verification is also important to guarantee that code compiled from legal Java programs is accepted and executed by the JVM.

The official specification of the JVM [3], which includes bytecode verification, is written in informal English prose. Most of this specification is rather clear, but there are some inadequacies. Their presence is particularly problematic for security-critical features such as bytecode verification, because erroneous interpretation can lead to erroneous implementation.

This paper exposes and analyzes the inadequacies in the official specification of bytecode verification, providing concrete suggestions for improvement. All the inadequacies that the author is currently aware of are covered. Certain aspects of bytecode verification, which are directly relevant to the analysis and improvement of its specification, are extensively treated in other papers; this paper only provides an overview of such aspects, referring the reader to those other papers for details.

The next section clarifies the role of bytecode verification in the JVM, also identifying which parts of the JVM specification describe bytecode verification. The analysis and the suggested improvements are presented in Sections 3, 4, and 5, which correspond to the three parts of the specification. Related work is discussed in Section 6. The official JVM specification is denoted by 'JS'; individual chapters or (sub-)sections are denoted by appending their number, e.g. 'JS3.1'.

The goal of this paper, like others in the field, is to improve the understanding, assurance, and usability of Java. The need and desire to improve JS, in particular the description of bytecode verification, 'ideally to the point of constituting a formal specification', is explicitly stated in the Appendix of JS; this paper contributes to that goal.

## 2. ROLE OF BYTECODE VERIFICATION

### 2.1. Class file verification

JS4 describes the format of valid class files. Class files may come from a variety of sources, such as the local file system or a network connection; they may even be constructed or instrumented on the fly. Ultimately, they are presented to the core class creation mechanisms of the JVM as byte sequences, from which the JVM creates (internal representations of) classes and interfaces. Class file verification is the process of checking that a byte sequence constitutes a valid class file.

Class file verification is described in JS4.9.1 as consisting of four passes. Passes 1 and 2 check the format of the class file, excluding those byte subsequences that constitute methods' code. The boundaries of such subsequences are identified during passes 1 and 2, but it is pass 3's responsibility to verify that each of them constitutes valid code for an individual method. Pass 4 consists of the checks performed by resolution, i.e. that symbolic references from instructions to classes, interfaces, fields, and methods are correct.

Bytecode verification is pass 3, which is the most interesting and delicate one. The other passes are relatively straightforward and do not present major difficulties.

### 2.2. Goal

The purpose of bytecode verification is checking whether a byte sequence constitutes valid code for a method. The checking is performed with respect to contextual information that includes the method's signature and the constant pool of the class file.

An instruction consists of a one-byte opcode, which specifies an operation to be performed, followed by zero or more bytes encoding operands, which specify data to be operated upon. Bytecode verification must preliminarily check that the byte sequence constitutes an instruction sequence (i.e. correct opcodes, etc.). Then, it must establish that the instruction sequence satisfies certain type safety properties.

The exact type safety properties to establish can be determined from the specification of instructions in JS6. That specification includes statements using 'must', such as 'the top of the operand stack *must* contain an integer'. As explained in JS6.1, the meaning of 'must' is that the execution engine expects the expressed requirements to hold, and it is the task of class file verification to make sure they indeed hold[§].

Some of these requirements are checked by the resolution process, i.e. pass 4 of class file verification. An example is that the method symbolically referenced by `invokevirtual` must exist, have the indicated argument and return types, and be accessible from the class where `invokevirtual` occurs.

Passes 1 and 2 check certain 'implicit' requirements. An example is that the constant pool must be well-formed. Another example is that there must be no circularities in the type hierarchy; this check involves not only the class file under verification, but also the classes and interfaces currently loaded in the JVM.

The remaining requirements are checked by bytecode verification, i.e. pass 3. An example is that the index of a local variable used as an operand must be within the range of the local variables of the method. Another example is that the top of the operand stack must contain an integer when certain instructions are executed.

If the rectangular space in Figure 1 represents all possible code (i.e. all byte sequences), then the larger, full-line oval delimits code that is acceptable by bytecode verification. This consists of all byte

[§]There are a few exceptions to this statement, i.e. sentences in JS6 that use 'must' for checks to be performed at run time. For example, the specification of `aastore` states that the type of the object to store *must* be assignment compatible with the component type of the array. However, the same specification states that a run time exception is thrown if that is not the case. So, despite the use of 'must', it is clearly not the task of class file verification to check that property; this also applies to the other sentences in JS6 where 'must' is used for a run time check.

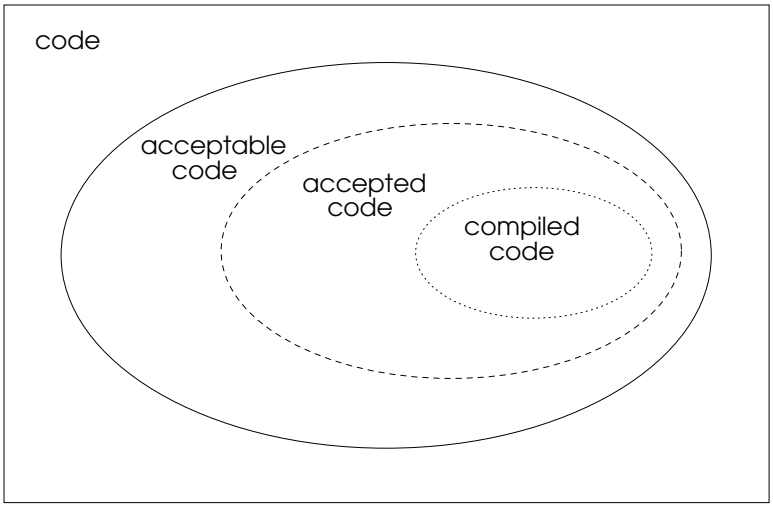



Figure 1. Requirements for bytecode verification.

sequences encoding instruction sequences that satisfy the type safety properties derived from JS6 that must be checked by pass 3.

Unfortunately, it is undecidable whether a byte sequence is acceptable code. As is well-known in program analysis, properties such as 'the top of the operand stack always contains an integer when a certain instruction is executed' can be checked only approximately. Bytecode verification is bound to be a decidable approximation of an ideal filter that exactly recognizes acceptable code: the mid-size, dashed-line oval in Figure 1 delimits the code that is actually accepted by bytecode verification.

As explicitly stated in JS4.9 and as mentioned in Section 1, code generated by (correct) compilers must be accepted by bytecode verification. The smaller, dotted-line oval in Figure 1 delimits code generated by compilers.

The containment relationship among the ovals must be as indicated in the figure. Any choice of the accepted code oval is fine, as long as it is contained in the acceptable code oval and contains the compiled code oval. Failure of either containment would cause type unsafety or rejection of legal Java programs.

While acceptable code can be characterized precisely from the specification of instructions in JS6, compiled code depends on how compilers are implemented. Even if all current compilers used common compilation strategies, future compilers might use different strategies to generate better code. The compiled code oval is a moving target, not very suitable to universal characterization.

The best approach is to give a precise characterization of accepted code. JVM implementors should write bytecode verifiers that exactly recognize the accepted code oval. The characterization also constitutes a contract with compiler developers: as long as a compiler generates code that falls inside the oval, that code will pass bytecode verification.

The definition of accepted code should embody an optimal trade-off between two criteria: (1) bytecode verification should be as simple and efficient as possible; and (2) it should accept as much code as possible. Privileging the first criterion may limit future compilers or reject some code from current compilers; privileging the second may make implementations more susceptible to errors and hence to attacks exploiting the errors.

### 2.3. Specification

Bytecode verification is described in JS4.8 and JS4.9. JS4.8 presents constraints that a byte sequence must satisfy in order to represent valid method code; they are divided into static and structural constraints. JS4.9 explains, in a quite algorithmic way, how such constraints are checked.

## 3. STATIC CONSTRAINTS

Static constraints, presented in JS4.8.1, state the properties that a byte sequence must satisfy in order to constitute an instruction sequence with correct opcodes and operands. All the instructions' operands are covered. The constraints are quite straightforward. Some involve the constant pool; for instance, the operand of `getfield` must point to a field reference in the constant pool.

One constraint seems out of place, though. It states that `new` cannot be used to create an instance of an interface or of an abstract class.

The constraint must indeed hold, but it is given in the context of restrictions on `new`'s operand. Since `new` references a class by name (via an index into the constant pool), the name should be resolved in order to determine whether it denotes an interface, an abstract class, or a non-abstract class. Unlike all the other static constraints, this one cannot be checked locally to the class file.

Furthermore, the specification of `new` in JS6 states that an exception is thrown if the name resolves to an abstract class or an interface. This means that the check is performed at resolution time. Other instructions that reference classes, interfaces, fields, and methods in their operands require checks similar to this one (e.g. the field referenced by `getfield` must not be static); they are given as resolution time checks in JS6, not as static constraints.

For these reasons, the constraint on `new` described above should be removed from JS4.8.1.

Even though static constraints are derivable from JS6, it seems indeed useful to have all of them collected in one place of JS, namely JS4.8.1.

## 4. STRUCTURAL CONSTRAINTS

Structural constraints, presented in JS4.8.2, state (for the most part) type safety properties that must hold when instructions are executed. These constraints apply to instruction sequences; they assume that static constraints are satisfied.

### 4.1. Terminology

Both static and structural constraints are meant to be checked *statically*, prior to executing the code. Using the adjective 'static' only for the first kind of constraints sounds a little misleading and confusing.

The terms ‘static’ and ‘structural’ could be replaced by something like ‘well-formedness’ and ‘typing’, respectively. However, in the rest of this paper the adjectives ‘static’ and ‘structural’ will still be used.

### 4.2. Undecidability

Most structural constraints state properties that must hold when certain instructions are executed. For example, when `getfield` is executed, the top of the operand stack must contain a reference to an instance of a class that is assignment compatible with the class or interface referenced by the operand. As mentioned in Section 2.2, precisely establishing this kind of property before execution is undecidable. So, structural constraints define acceptable code, as opposed to accepted code (using the terminology introduced in Section 2.2). Accepted code is defined by the algorithm described in JS4.9. This point should be stated explicitly in JS4.8.2.

### 4.3. Redundancy

The first constraint in the list states that each instruction must be executed with appropriate number and types of values in the operand stack and local variables. The notion of ‘appropriate’ can be determined from JS6. For example, `putfield` requires the value stored in the field to be assignment compatible with the type of the field.

The same requirement on `putfield` is stated explicitly by a separate constraint. Since it can be derived from the first one, it is redundant.

Another constraint states that no instruction must pop more values from the operand stack than it contains, i.e. no stack underflow must occur. Again, this is a simple consequence of the first constraint in the list; each instruction requires the presence of a certain number of values in the operand stack, and only those values are popped.

It could be argued that even the first constraint in the list is redundant. The specification of instructions in JS6 includes the types of the values that each instruction expects to find in the operand stack and local variables. As explained in JS6.1, it is the task of bytecode verification to ensure that these expectations are met, i.e. that each instruction is executed with appropriate number and types of values in the operand stack and local variables.

Despite this redundancy, it seems useful to have all the structural constraints collected in one place, namely JS4.8.2. Unlike static constraints, which cover all the instructions’ operands, the structural constraints currently present in JS4.8.2 do not cover all the instructions’ typing requirements. Constraints for the missing instructions should be added. The first constraint, which subsumes the others, should be eliminated or used as a summary statement before the list.

### 4.4. Unexplained restrictions

#### *4.4.1. Uninitialized objects*

A constraint states that no (reference to an) uninitialized object must be present in the operand stack or local variables when a backward branch is taken, or in a local variable in code protected by an exception handler. These requirements have no direct bearing on type safety: for type safety: it is sufficient that

uninitialized objects are initialized before they are used, regardless of their presence when backward branches are taken or in code protected by exception handlers.

JS4.8.2 provides no explanation for the constraint, which turns out to be a 'forward reference' to the algorithm described in JS4.9, which statically checks that the constraint will be satisfied at run time; see Section 5.8.2. Since structural constraints define acceptable code and not accepted code, the constraint should be removed from JS4.8.2.

### *4.4.2. Subroutines*

Some constraints state requirements on subroutines: subroutines cannot be called recursively; the instruction following a `jsr` may be returned to only by a single `ret`; each return address may be returned to at most once; etc. These requirements have no direct bearing on type safety, for which it is sufficient that `jsr` does not cause an operand stack overflow and that a return address is present in the local variable referenced by `ret`.

No explanation for these constraints is given in JS4.8.2. Like the one analyzed in Section 4.4.1, these constraints are 'forward references' to the algorithm described in JS4.9; see Section 5.9.2. For the same reason given in Section 4.4.1, they should be removed from JS4.8.2.

### *4.4.3. Operand stack size*

A constraint states that if an instruction can be executed along different paths, the operand stack must always have the same size prior to the execution of the instruction, regardless of the path taken. Again, this requirement has no direct bearing on type safety: as long as the stack does not exceed the maximum size for the method, it may have different sizes when a certain instruction is executed.

Again, no explanation is given in JS4.8.2, but the following explanation can be gathered from [6].

If the constraint is satisfied, every instruction always accesses the same elements of the stack when it is executed. For instance, if the instruction is `iadd` and the stack size is always 3, the instruction always reads the integers from the third and second elements of the stack (counting from the bottom) and pushes their sum into the second element.

This invariant allows, in typical microprocessors, the stack elements to be implemented as native registers, which are accessed directly. For instance, the `iadd` considered in the example above reads the integers from the registers for the third and second stack elements, and writes their sum into the register for the second stack element; in typical microprocessors, this operation is realized by a single native machine instruction.

In other words, the constraint supports more efficient execution of bytecode instructions by means of native machine instructions (e.g. resulting from just-in-time compilation). If the constraint were not satisfied, the stack would have to be implemented explicitly with a current size that is incremented and decremented, resulting in poorer performance.

This motivation for the above constraint is not totally obvious. In addition, it is irrelevant to certain architectures where the JVM could be realized. For example, in a hardware JVM that directly executes bytecode instructions, stack manipulation operations could be micro-coded and hence as fast as register manipulation operations. This is not to say that the constraint should necessarily be eliminated, because

its presence does not cause any problem in any architecture; some architectures may take advantage of it. Anyhow, at least some explanation should be added to JS4.8.2.

#### *4.4.4. Special method invocations*

A constraint requires `invokespecial` to reference an instance initialization method or a method in (a superclass of) the current class. Again, this requirement has no direct bearing on type safety: if `invokespecial` references a non-instance initialization method that is not in (a superclass of) the current class, the method is directly invoked, as specified in JS6; the method's class is (a superclass of) the target class, as ensured by bytecode verification.

No explicit explanation is given in JS4.8.2. Some explanation can be derived by considering the circumstances in which compilers generate `invokespecial`. This instruction is used to compile calls to constructors, calls to private methods, and calls with target `super`, for which (the typical implementation of) the lookup procedure of `invokevirtual` is inappropriate. As a consequence of this compilation strategy, `invokespecial` in compiled code always calls an instance initialization method or a method in (a superclass of) the current class. Despite this compilation strategy, the above constraint is irrelevant to type safety and should be removed from JS4.8.2.

### 4.5. Contradiction

Two constraints slightly contradict each other. One states that the fields of an uninitialized object cannot be accessed; the object must be initialized first. The other states that the code of an instance initialization method can store values in the fields of the object to be initialized, which, strictly speaking, is still uninitialized.

Admittedly, this is not a very severe contradiction; the former constraint can be viewed as a broader statement that is relaxed by the latter. However, the two statements could be re-worded, and perhaps merged together, to clarify that the latter relaxes the former.

### 4.6. Incorrect wording

A constraint states the following requirement for `getfield`, `putfield`, `invokevirtual`, and `invokespecial`: if the referenced field or method is protected and is declared in a superclass of the current class, then the class of the object whose field or method is being accessed must be either the current class or a subclass of it. This requirement derives from an analogous requirement for Java [2]. It has to do with the principle that the protected members (and constructors) of an object may be accessed from outside their package only by code that is responsible for the implementation of that object.

The wording of the constraint is incorrect. A protected member is always accessible from within the package where the member is declared, without restrictions; the requirement described above only applies if the member is declared outside the package of the current class [2]. The constraint should be re-written as follows: if the referenced member is protected and is declared in a superclass of the current class that is in a different package, then the class of the object whose member is being accessed must be either the current class or a subclass of it.

A similarly incorrect wording is present in the specifications of `getfield`, `putfield`, `invokevirtual`, and `invokespecial` in JS6; it should be similarly corrected.

## 4.7. Possibly static constraints

### *4.7.1. Method return instructions*

A constraint requires each method return instruction (such as `ireturn` and `freturn`) to match the method's return type. This is a simple check which does not involve any type analysis. So, it could well be moved into JS4.8.1, becoming a static constraint. Of course, type analysis is required to establish that the operand stack always has a value of the right type when the instruction is executed. However, this is a separate constraint.

### *4.7.2. Last instruction in the sequence*

A constraint states that execution must never fall off the bottom of the code. This is also a simple check that does not involve any type analysis; the last instruction of the sequence must be one that cannot transfer control to the non-existent following one, directly or indirectly[¶]. In addition, some static constraints require the operands of the control transfer instructions to point to instruction addresses, and not outside the code or in the middle of an instruction; the requirement that execution cannot fall off the end of code is definitely related to these requirements. For these reasons, the constraint above could well become a static constraint too.

## 4.8. Heterogeneity

The structural constraints currently present in JS4.8.2 are somewhat heterogeneous. Most of them state typing requirements that must hold when certain instructions are executed. Some state requirements relating all possible executions, e.g. that the instruction following a `jsr` may be returned to by a single `ret`. Others state requirements on instructions' occurrences, e.g. that return instructions must match the method's return type.

Heterogeneity is not necessarily bad, but it contrasts with the homogeneity of static constraints, which are all local checks on instruction opcodes and operands. However, if the suggestions in Sections 4.4 and 4.7 to remove constraints from JS4.8.2 were followed, then the remaining structural constraints would be quite homogeneous, all stating typing requirements that must hold when certain instructions are executed; this is also true for the constraints that Section 4.3 suggests to add to JS4.8.2. The only mild exception would be the constraint on the operand stack size analyzed in Section 4.4.3, which relates all possible executions.

[¶]The 'indirect' case applies to `jsr`: control is transferred to the following instruction when returning from the called subroutine. If `ret` does not occur in the code, then it is impossible to return to the instruction following a `jsr`. So, `jsr` could be allowed as the last instruction if `ret` does not occur in the code. However, a method with `jsr` as the last instruction and without `ret` is definitely pathological. Thus, not much is lost by requiring the stronger condition that the last instruction cannot be `jsr`, regardless of the presence of `ret` in the code.

## 5. VERIFICATION ALGORITHM

JS4.9 describes an algorithm for bytecode verification. The initial part of JS4.9.2 essentially explains how to check static constraints and is quite straightforward; a mildly interesting point is the inclusion of the check that execution cannot fall off the end of code, supporting the view of this requirement as a static constraint, as argued in Section 4.7.2. The remaining part of JS4.9.2 explains how to check a decidable approximation of structural constraints, by means of a data flow analysis [7]. Clarifications concerning the treatment of certain instructions are given in JS4.9.3 through JS4.9.6.

### 5.1. Ambiguity of reference types

As stated in JS5.3, a class or interface in the JVM is identified by its (fully qualified) name plus its defining class loader. However, classes and interfaces are symbolically referenced in class files by name only. The description of the data flow analysis talks about 'reference types' that are assigned to the operand stack and local variables, but it fails to clarify whether these reference types consist of names only or names plus class loaders.

Since class loaders are run time objects that may be user-defined, the only way to know their identities is to actually load the classes and interfaces. This counters the statement in JS2.17.1 that classes and interfaces may be loaded lazily, when they are required for execution. Thus, the most reasonable interpretation is that the reference types used by the algorithm consist of names only.

This is correct as long as there is an intended disambiguation of names, accompanied by mechanisms to ensure consistent disambiguation between methods that exchange objects. Consider the following example [5,8]. A method $m_1$ has an argument of type `C`, whose disambiguation in $m_1$ is a class $c$ (identified by the name `C` plus some loader). Another method $m_2$ calls $m_1$, passing an object of type `C` to it. If `C` is disambiguated to a class $c'$ in $m_2$, it must be $c = c'$; otherwise, type safety could be broken.

In the first edition of JS, these issues were not mentioned. Type safety bugs due to inconsistent disambiguation of names were found in earlier implementations of the JVM [8]. Those bugs were corrected by the introduction of loading constraints [9], described in JS5. Loading constraints ensure that classes exchanging objects agree on the actual classes of the exchanged objects, not only on their names. Loading constraints are external to bytecode verification; they are part of the class loading mechanisms, which complement bytecode verification to ensure type safety, together with resolution and other mechanisms.

Formal evidence that bytecode verification can use names only and rely on loading constraints to ensure their consistent disambiguation is given in [10]. A name `N` used in a class $c$ stands for the class or interface $\mathtt{N}^L$, where $L$ is the defining loader of $c$. The notation $\mathtt{N}^L$, used in JS5, denotes the class or interface with name `N` and initiating loader $L$; the loaded class cache and the class creation mechanisms of the JVM ensure that $\mathtt{N}^L$ denotes a unique class or interface. A loading constraint has the form $\mathtt{N}^L = \mathtt{N}^{L'}$, expressing the fact that the two classes or interfaces with name `N` and initiating loaders $L$ and $L'$ (with $L \neq L'$) must be the same class or interface.

JS4.9.2 should explicitly state that the reference types used by the algorithm consist of names only and that the intended disambiguation of a name `N` is the class or interface $\mathtt{N}^L$, where $L$ is the defining loader of the class under verification.



### 5.2. Subtype checking

During bytecode verification, a class or interface type C in the operand stack may be the target of a member access instruction whose operand references D as the class or interface of the member to be accessed. While the existence of the member is checked when the reference to the member is resolved, bytecode verification must ensure that C is a subtype of D. According to JS4.9.1, this is done by resolving the two names and checking that the required subtype relation holds between the actual classes or interfaces[‖]. This strategy results in premature loading.

A better approach is to generate a subtype constraint of the form $\mathtt{C}^L < \mathtt{D}^L$, where $L$ is the defining loader of the class under verification [10,11]. Subtype constraints can be checked lazily when classes and interfaces are loaded, analogously to the equality constraints introduced in [9]. Formal evidence that type safety is guaranteed is given in [10].

There is nothing wrong with resolving names and checking the required subtype relation on the actual classes or interfaces. However, the generation of subtype constraints is cleaner and allows lazier loading. Thus, subtype constraints could well be added to JS.

### 5.3. Merging of reference types

JS4.9.2 prescribes that the result of merging two class types C and D is their first common superclass. This requires resolving the two names to actual classes, and then traversing their ancestry to find the first common superclass.

The interaction of this strategy with subtype checking causes a type safety bug [11,12], unless extra precautions are taken. Suppose that the result of merging C and D is S and that this name is checked against the name S referenced by a member access instruction: the check succeeds because the names are equal. However, the first common superclass of $\mathtt{C}^L$ and $\mathtt{D}^L$ (where $L$ is the defining loader of the class under verification), despite having name S, may have a different defining loader from the class $\mathtt{S}^L$ whose member is accessed. See [11] for details.

The problem is that this merging strategy may break the property that a name N used in bytecode verification stands for the class or interface $\mathtt{N}^L$, where $L$ is the defining loader of the class under verification. So, bytecode verification cannot use names only. Additional information is needed to properly disambiguate names obtained by merging and to make correct type comparisons. The additional information could be the defining loader of the first common superclass that results from merging [11].

This problem is not mentioned in JS, but can be easily overlooked. For example, Sun's Java 2 SDK version 1.4 is affected by this bug. The problem and its solution should be discussed in JS.

A better strategy is to assign finite sets of reference types to the operand stack and local variables, merging the sets by union [11,13,14]. For example, the result of merging {C} and {D} is {C, D}. This avoids the above problem altogether, because no 'new' name is introduced by merging: a name N

[‖]Subtype checking is described in JS4.9.1, in the context of lazy loading. Indeed, the description could be moved or copied into JS4.9.2, because it is an important part of the data flow analysis.

always stands for the class or interface $\mathtt{N}^L$. In addition, no classes need to be loaded for merging. Given its simplicity and advantages, this strategy could well be incorporated into JS.

### 5.4. Interface types

The first edition of JS prescribes that the result of merging two reference types is their first common superclass or superinterface. This works fine for classes, but the first common superinterface of two interfaces may not be unique because of multiple inheritance. In the second edition of JS the statement has been changed to say that the result of merging two reference types is just their first common superclass. Since `java.lang.Object` is considered a superclass of every interface, the result of merging two interface types is always `java.lang.Object`.

This strategy requires a special treatment of `java.lang.Object` when it is the target of `invokeinterface`. Since `java.lang.Object` may derive from merging two interface types, bytecode verification should allow `invokeinterface` to operate on it. Otherwise, some compiled code would be rejected. Normally, `invokeinterface` should be only allowed to operate on a subtype of the interface referenced by `invokeinterface`, but `java.lang.Object` is not a subtype of any interface. In order to maintain type safety, a run time check is necessary when `invokeinterface` is executed.

This treatment of interface types is not particularly clean. Furthermore, JS does not make its implications entirely clear, e.g. that `java.lang.Object` needs a special treatment. The specification of `invokeinterface` in JS6 mentions the needed run time check (by saying that an exception is thrown if the class of the target object does not implement the interface), but the relation between this run time check and bytecode verification is not made explicit. These implications should be discussed in JS.

The alternative merging strategy described in Section 5.3 automatically provides a clean and sound treatment of interface types [11,13,14]. Interface names are treated exactly like class names**. Since merging is set union, multiple inheritance of interfaces is not a problem; there is no need to use `java.lang.Object` as the result of merging. Therefore, no special treatment of `java.lang.Object` is necessary and no run time checks need to be performed when `invokeinterface` is executed. This is one more reason why this merging strategy could be incorporated into JS.

### 5.5. Special names

Certain names denote system classes and interfaces that play special roles in the type hierarchy or in class files: `java.lang.Object` denotes the root of the class hierarchy; `java.lang.String` denotes the class of the strings in the constant pool referenced by `ldc` and `ldc_w`; `java.lang.Throwable` denotes the root of all classes whose instances can be thrown

**In general, whether a name denotes a class or an interface can be only determined by resolving the name. For instance, if a method has an argument of reference type `R`, that name may denote a class as well as an interface.

as exceptions; and all arrays implement the interfaces denoted by `java.lang.Cloneable` and `java.io.Serializable`.

Since a class or interface is identified by its name plus its defining loader, nothing prevents the existence of user-defined classes and interfaces with these 'special' names and different loaders. JS1.4 says that the names in the package `java` that are used in JS denote the classes and interfaces as loaded by the bootstrap class loader, i.e. the expected system classes and interfaces. However, JS does not explicitly require a JVM implementation to forbid the existence of user-defined classes and interfaces with the special names.

So, bytecode verification cannot always assume that, e.g. the name `java.lang.Object` denotes the root of the hierarchy: such a name could resolve to a user-defined class. If bytecode verification considered any reference type `R` to be a subtype of `java.lang.Object`, type safety could be broken. It is necessary either to check that $\texttt{java.lang.Object}^L$ (where $L$ is the defining loader of the class under verification) is the root of the class hierarchy or to treat the name `java.lang.Object` as a regular one [11].

This point is not discussed in JS, but can be easily overlooked. In Sun's Java 2 SDK version 1.2, type safety can be broken by means of user-defined classes with special names [11]. This bug has been corrected in later versions of Sun's Java 2 SDK by forbidding the existence of user-defined classes or interfaces in the `java` package. JS should either state that user-defined classes or interfaces with the special names are disallowed or discuss the implications for bytecode verification.

### 5.6. Protected fields and methods

A subtle and often neglected aspect of bytecode verification is the treatment of protected members. As mentioned in Section 4.6, if the member accessed by an instruction is protected and is declared in a superclass of the current class that is in a different package, then the class of the object whose member is being accessed must be either the current class or a subclass of it. This requirement should be somehow checked by bytecode verification, to avoid expensive run time checks.

A straightforward approach is to resolve the reference to the member and see if the member is protected and declared in a superclass of the class under verification that belongs to a different package. If that is the case, the class in the operand stack is checked to be the class under verification or a subclass of it.

In certain cases, the requirement can be soundly checked by just inspecting the superclasses of the class under verification. For example, if no superclass declares a protected member with the given name and descriptor, the requirement is certainly satisfied [15]. Inspection of the superclasses does not cause any additional loading because when a class is loaded all its superclasses are also loaded, as specified in JS5.3.5. However, in general it is necessary to resolve the reference to the member, which may cause loading.

An approach that allows lazier loading is to generate a conditional subtype constraint [15] of the form **if** $\mathit{ProtCond}(\texttt{S}^L.\texttt{n}{:}d, \texttt{C}^L)$, **then** $\texttt{D}^L < \texttt{C}^L$ where `C` is the name of the class under verification, $L$ its defining loader, `S` the name of the referenced class, `n` the name of the referenced member, $d$ its descriptor, and `D` the class name in the operand stack (where $\texttt{D} \neq \texttt{C}$; no constraint needs to be generated if $\texttt{D} = \texttt{C}$). The notation $\texttt{S}^L.\texttt{n}{:}d$ denotes the member with name `n` and descriptor $d$ in class $\texttt{S}^L$ (if present; if not, the first member with name `n` and descriptor $d$ found in the superclasses of $\texttt{S}^L$, according to the definition of field and method resolution in JS5.4.3.2 and JS5.4.3.3). The predicate

*ProtCond*$(m, c)$ holds on a member $m$ and a class $c$ when $m$ is declared in a superclass of $c$, $m$ is protected, and $m$ and $c$ belong to different packages.

JS should discuss the treatment of protected members; conditional subtype constraints could be incorporated.

### 5.7. Purely functional interface

By generating (unconditional and conditional) subtype constraints and by using sets of reference types that are merged by union, bytecode verification never needs to load any class or interface and can be made a purely functional component of the JVM. It is activated by passing a byte sequence as input, purported to be method code, accompanied by contextual information such as the method's signature and the constant pool. The returned answer is either failure or success; in the latter case, a set of (unconditional and conditional) subtype constraints is also returned.

This is in spirit with the statement in JS2.17.1 that classes and interfaces may be loaded as lazily as desired, and with the statement in JS4.9.1 that bytecode verification avoids loading classes and interfaces unless it has to. JS does not *require* classes and interfaces to be loaded as lazily as possible; it *allows* them to be loaded as lazily as desired. Eager loading amounts to subtype constraints being checked as soon as they are generated. So, the purely functional interface, while increasing the potential laziness and the conceptual cleanness, does not preclude eager loading.

It turns out that people at Sun had independently devised subtype constraints, and implemented an experimental verifier that generates such constraints [16]. However, they decided not to change JS and their official verifier to avoid introducing more potential indeterminacy in the behavior of the JVM. Since loading classes and interfaces may have visible effects, especially with user-defined class loaders, certain programs may behave differently in implementations of the JVM that load classes and interfaces with different laziness. This may also happen without subtype constraints, though, because resolution may take place more or less lazily, as stated in JS2.17.1; subtype constraints would increase the potential indeterminacy only slightly.

### 5.8. Object initialization

#### *5.8.1. Basic approach*

Bytecode verification must ensure that objects are initialized before they are used. JS4.9.4 prescribes that the data flow analysis uses a special type for (references to) newly created but still uninitialized objects. This 'uninitialized type' is changed to a regular class type when an instance initialization method is invoked. Since several copies of the uninitialized object may be present in the operand stack and local variables when the instance initialization method is invoked, all the occurrences of the uninitialized type must be changed.

After an object is created and before it is initialized, another object of the same class may be created. The data flow analysis must distinguish between types for the first object and types for the second object. The strategy prescribed in JS4.9.4 is to index the type for an uninitialized object with the address of the `new` that creates it. Since the two objects are created at different addresses, there will be different types associated to them.

### 5.8.2. *Unnecessary checks*

A potential source of trouble is the occurrence of `new` inside a loop. If the loop does not initialize the object, how can the data flow analysis distinguish between objects created during two different iterations through the loop? To avoid this problem, JS4.9.4 requires that no uninitialized type exists in the operand stack or local variables when a backward branch is taken. For analogous reasons, no uninitialized type must exist in a local variable in code protected by an exception handler (no restrictions are given on the operand stack because the stack is emptied when an exception is thrown; the thrown exception is then pushed onto the stack).

This provides some explanation for the structural constraint mentioned in Section 4.4.1, but these restrictions on (types for) uninitialized objects are completely unnecessary not only as structural constraints, but also in the data flow analysis.

Suppose that `new` occurs at address $i$. Consider a path from address 0 (i.e. the start of the method's code) to $i$, such that the path does not include $i$ (except at the end)[††]. Clearly, no uninitialized type is assigned to the operand stack or local variables at address 0. The data flow analysis propagates types from 0 to $i$, transforming them according to the instructions encountered along the path. Since none of those instructions has address $i$, no uninitialized type with index $i$ can appear at $i$; it appears at the instruction following $i$. Even if there are circular paths from $i$ back to $i$, all the copies of the uninitialized type with index $i$ are eventually merged with the types at $i$, thus disappearing. Therefore, no confusion can arise: an uninitialized type with index $i$ always refers to the last object created by the `new` at address $i$.

Consider, e.g. the code in Figure 2[‡‡]. Types are propagated from 0 to 3; no uninitialized type appears at these instructions. The `new` instruction pushes onto the operand stack the uninitialized type $\mathsf{uninit}[\mathtt{C}]_3$. This type is moved into variable 2, where it stays throughout instructions 5, 6, and 7. The backward branch from 7 to 2 causes the type $\mathsf{uninit}[\mathtt{C}]_3$ to be merged with whatever type is present in variable 2 at instructions 0, 1, and 2, which is certainly a different type: the result is the type $\mathsf{undef}$, denoting an unusable or undefined type. The object initialized by `invokespecial` is the one created in the last iteration through the loop; those created during the previous iterations may still exist (if garbage collection has not already destroyed them), uninitialized but inaccessible.

It can be formally proved that the restrictions on uninitialized types required in JS4.9.4 are indeed unnecessary [17]. So, they should be removed from JS.

### 5.8.3. *Interaction with subroutines*

Consider the code in Figure 3, adapted from [18]. The subroutine creates a new object but invokes no instance initialization method. The subroutine is called twice; the uninitialized object is saved into

[††] If there is no such path, then $i$ is statically unreachable and hence irrelevant.

[‡‡] For the sake of readability, a few licenses are taken in this and other bytecode listings. First, instruction addresses are consecutive numbers instead of code offsets. Accordingly, the operands of control transfer instructions are replaced by the addresses of their target instructions. Moreover, the symbolic references in the constant pool are embedded as operands in place of the constant pool indices.

```
 0: iconst_0
 1: istore_1
 2: iinc 1 1
 3: new C
 4: astore_2
 5: iload_1
 6: iconst_3
 7: if_icmpne 2
 8: aload_2
 9: invokespecial C/<init>()V
10: return
```

Figure 2. Objects created but not initialized inside a loop.

```
 0: jsr 9
 1: astore_1
 2: jsr 9
 3: astore_2
 4: aload_2
 5: invokespecial C/<init>()V
 6: aload_1
 7: getfield C/f I
 8: return
 9: astore_0
10: new C
11: ret 0
```

Figure 3. Interaction between object initialization and subroutines.

variable 1 the first time and variable 2 the second time. Since `invokespecial` initializes only the second object, `getfield` accesses an uninitialized object.

In the data flow analysis, variable 1 has type $\mathsf{uninit}[\mathtt{C}]_{10}$ at address 2. As described in Section 5.9.1, if a local variable is not modified inside a subroutine its type at a return address is propagated from the matching `jsr`. Since variable 1 is not modified inside the subroutine, its type $\mathsf{uninit}[\mathtt{C}]_{10}$ is copied from address 2 to 3, where the top of the operand stack has also type $\mathsf{uninit}[\mathtt{C}]_{10}$. At address 4 both variables 1 and 2 have type $\mathsf{uninit}[\mathtt{C}]_{10}$, which is uniformly changed to a regular type by `invokespecial`. The access by `getfield` is then erroneously deemed legal.

The problem is that types for previously created but still uninitialized objects can be propagated to a return address from the matching `jsr`, invalidating the property that a type with index $i$ is always associated to the last object created by the `new` at address $i$ [18]. This subtle point is not mentioned in JS4.9.4, but can be easily overlooked, leading to incorrect implementations. For instance, Sun's JDK version 1.1.4 incorrectly accepts the code in Figure 3 [18]; this bug has been corrected in later versions.

A draconian solution is to prohibit uninitialized types from appearing at any `jsr` or `ret`, i.e. no uninitialized object can enter or exit a subroutine [18].

A more targeted solution is just to prevent uninitialized types from being propagated to a return address from its matching `jsr`. This is realized by changing to undef the uninitialized types present at the `jsr` when they are propagated to the return address. In Figure 3, instead of propagating uninit$[\mathtt{C}]_{10}$ for variable 1 from address 2 to 3, undef is assigned to variable 1 at address 3. This causes the `aload` at address 6 to fail, rejecting the code. In other words, the previously created object in variable 1 is made unusable by using type undef for it. The soundness of this approach can be formally proved [17].

The interaction between object initialization and subroutines should be discussed in JS, along with the targeted solution just described.

## 5.9. Subroutines

### *5.9.1. Tracking modified variables*

As explained in JS4.9.6, compilers may generate code where a local variable holds values of different types (including undef) during different invocations of a subroutine. This means that the data flow analysis assigns undef to the variable in the subroutine. If the variable is not modified inside the subroutine, its type at each return address should be the same as the one at the matching `jsr`. However, a simple-minded data flow analysis would propagate undef from `ret` to the return address, rejecting some mundane compiled programs.

To avoid this problem, JS4.9.6 prescribes to keep track of the local variables (potentially) modified inside each subroutine. This information is used to propagate types for local variables to return addresses: if a variable is marked as modified, the type from `ret` is propagated; otherwise, the type from `jsr` is propagated.

### *5.9.2. Disciplined use of return addresses*

Consider the code in Figure 4, adapted from [19]. The subroutine is called twice. Variable 0 contains an object of type `D` the first time, an integer the second time. Variable 1 contains integer 0 the first time, integer 1 the second time. The subroutine tests the integer in variable 1: if it is 0 (i.e. at the first call) it saves the return address 5 into variable 2 and returns; otherwise (i.e. at the second call), it discards the new return address and returns to the previously saved one. After the second call, the integer in variable 0 is used as an object reference by `putfield`.

In the data flow analysis, variable 0 is not modified inside the subroutine; if the type `D` for variable 0 were propagated from address 4 to 5, the program would be unsoundly accepted. In other words, bytecode verification could be fooled by a devious program that makes an undisciplined use of return addresses.

Actually, the program in Figure 4 is readily rejected by any data flow analyzer because there is a path from address 0 to 19 (i.e. 0–4, then 14–15, then 18–19) that does not store any value into variable 2. So, variable 2 does not have a return address type at address 19, causing `ret` to fail. Nonetheless, it is possible to construct more involved programs that can fool data flow analyzers that do not ensure some discipline in the use of return addresses; see [20] for some examples.

This danger is likely to be the reason behind the unexplained structural constraints about subroutines mentioned in Section 4.4.2. These constraints are meant to ensure some discipline in the use of return

```
 0: getstatic C/f LD;
 1: astore_0
 2: iconst_0
 3: istore_1
 4: jsr 14
 5: aload_0
 6: iconst_0
 7: putfield D/g I
 8: iconst_3
 9: istore_0
10: iconst_1
11: istore_1
12: jsr 14
13: return
14: iload_1
15: ifne 18
16: astore_2
17: ret 2
18: pop
19: ret 2
```

Figure 4. Undisciplined use of return addresses.

addresses, in order to avoid type safety problems. Since they concern the verification algorithm, they should be moved into JS4.9.6.

Subroutines can be nested, e.g. in code generated from nested `finally` blocks. JS4.9.6 prescribes to maintain, for each instruction address, a list of all subroutines (i.e. their starting addresses) needed to reach that address. For each subroutine in the list, information is kept about the local variables modified since the subroutine was called.

This may deceptively seem a simple strategy, easy to realize in a data flow analyzer. The `jsr` instruction extends the list, first checking that the called subroutine is not in the list; this check prevents recursive subroutine calls, as required by one of the structural constraints. The `ret` instruction shrinks the list and uses the information about modified variables to propagate types from `ret` or `jsr`.

### *5.9.3. Complications*

Consider the Java program in Figure 5, adapted from [21]. The bytecode for method `m` is shown in Figure 6; variable `b` in the Java source is mapped to variable 0 in the bytecode. The subroutine generated from the `finally` block can be exited via the branch generated from the `continue` statement [21].

A subroutine may also be exited via an exception, i.e. control is transferred from the subroutine to a handler. This happens in bytecode generated from Java code where an exception is thrown inside a `finally` block that occurs inside a `try` block [22].

The possibility that subroutines are not exited via `ret` makes the treatment of the lists of subroutines less clear. It is not immediate for the data flow analysis to figure out whether a branch or thrown exception is exiting a subroutine or not. The check for recursive subroutine calls also becomes

```
class C {
    static void m(boolean b) {
        while (b) {
            try {
                b = false;
            } finally {
                if (b) continue;
            }
        }
    }
}
```

Figure 5. A subroutine that can be exited via branching.

```
 0: goto 14
 1: iconst_0
 2: istore_0
 3: jsr 9
 4: goto 14
 5: astore_1  // exception handler protecting addresses 1 to 4
 6: jsr 9
 7: aload_1
 8: athrow
 9: astore_2
10: iload_0
11: ifeq 13
12: goto 14
13: ret 2
14: iload_0
15: ifne 1
16: return
```

Figure 6. Bytecode for the method in Figure 5.

problematic: for instance, in the program in Figure 6, if the (singleton) list containing the (only) subroutine is propagated from address 12 to 14 and then back to 1, a recursive subroutine call is erroneously detected at address 3.

A related problem is how to merge, at converging control paths, the lists of subroutines and their associated information about modified variables. The case in which the lists to be merged contain the same subroutines in the same order is easy: for each subroutine, the modified variables are those modified in any path (i.e. the union is taken). However, if the lists have different sizes and/or subroutines, it is not obvious how to merge them.

So, the combination of the structural constraints about subroutines in JS4.8.2 and the strategy sketched in JS4.9.6 provide a blurry picture of how bytecode verification should treat subroutines. It is unclear which requirements are necessary for which purpose and how to check them (e.g. recursive subroutine calls). A detailed analysis of these requirements can be found in [20], where it is also shown that some of them are unnecessary to guarantee type safety.

```
class C {
    static int m(boolean b) {
        int i;
        try {
            if (b) return 1;
            i = 2;
        } finally {
            if(b) i = 3;
        }
        return i;
    }
}
```

Figure 7. A legal program rejected by most verifiers.

Since the involved subtleties are not obvious, the door is open to incorrect implementations where type safety can be broken and/or certain compiled programs are rejected. For example, the off-card verifier of Sun's Java Card Development Kit version 2.1.2, developed by Trusted Logic, rejects the bytecode in Figure 6 because of a false recursive subroutine call detected at address 3 [20,23].

### *5.9.4. Fundamental limit*

Consider the Java program in Figure 7, adapted from [24]. It is reported in [24] that the resulting bytecode is rejected by all the verifiers tried by the authors, including those in various versions of Sun's Java 2 SDK, Netscape, and Internet Explorer, as well as the Kimera verifier [25].

The bytecode for method `m` is shown in Figure 8; variables `b` and `i` in the Java source are respectively mapped to variables 0 and 1 in the bytecode. Since inside the subroutine there is a path that stores integer 3 into `i`, `i` is marked as modified by the subroutine. Its type at `ret` is thus propagated to return addresses 5, 10, and 13. Since there is a path from address 0 to 20 (through the call from address 4) that does not store any value in `i`, the type of `i` at `ret` is undef. This eventually causes the `iload` at address 21 to fail.

The Java program in Figure 7 is legal: variable `i` is definitely assigned a value before it is used, according to the rules of definite assignment of Java [2]. In the code in Figure 8, variable `i` is also definitely assigned a value before it is used; but the data flow analysis cannot establish that [24].

This example exposes a fundamental limit of the approach to subroutines described in JS. This limit is orthogonal to the unclear points of the description JS: in the example there is only one subroutine that is always exited via `ret`. The limit is inherent to the approach of tracking modified variables and selectively propagating types from `ret` and `jsr`.

### *5.9.5. Solution*

An alternative approach to subroutines that overcomes the fundamental limit exposed above is presented in [20,23]. The idea is the following; see [20,23] for details.

```
 0: iload_0
 1: ifeq 7
 2: iconst_1
 3: istore_2
 4: jsr 15
 5: iload_2
 6: ireturn
 7: iconst_2
 8: istore_1
 9: jsr 15
10: goto 21
11: astore_3  // exception handler protecting addresses 0 to 10
12: jsr 15
13: aload_3
14: athrow
15: astore 4
16: iload_0
17: ifeq 20
18: iconst_3
19: istore_1
20: ret 4
21: iload_1
22: ireturn
```

Figure 8. Bytecode for the method in Figure 7.

For each instruction address, instead of inferring just one assignment of types to the operand stack and local variables, the data flow analysis infers a set of such type assignments. Merging becomes set union, and the effect of an instruction is simulated element-wise on each assignment in the set, resulting in another set.

In addition, types for return addresses are indexed by return addresses, i.e. return addresses are isomorphic to their types. The `ret` instruction filters a set of assignments by propagating to a return address only those assignments that have the type of that return address in the variable referenced by `ret`. The `jsr` instruction pushes onto the operand stack the type for its matching return address.

There is no need to track the variables modified inside subroutines or to enforce any discipline in the use of return addresses; the treatment of `jsr` and `ret` is as simple as their run time behavior. The filtering of type assignments by `ret` achieves the effect of preserving the types of local variables (and operand stack as well) across subroutine calls that do not modify such variables.

The overhead of carrying around sets can be reduced by merging two or more type assignments into one when they do not have distinct return address types in the same location. This optimization does not result in a loss of precision because in that case the assignments would never be separated.

This approach is very easy to understand and implement as a data flow analysis that naturally generalizes the data flow analysis described in JS4.9.2. Furthermore, it is fully backward compatible. In fact, the set of accepted programs is very large; it has a simple characterization that arguably includes all code generable by present and future compilers [20,23].

Using sets of type assignments automatically achieves the same effect as using sets of reference types for single operand stack elements and local variables, explained in Section 5.3. However, the use of sets of reference types is still relevant in the context of the optimization to merge assignments that do not contain distinct return address types in the same location, mentioned above.

With this approach, the problematic interaction between subroutines and object initialization described in Section 5.8.3 can be easily eliminated. The idea is to index uninitialized types not with `new`'s addresses, but with consecutive natural numbers: when an object of class `C` is created, the type added to each assignment in the set is $\mathsf{uninit}[\texttt{C}]_i$, where $i$ is the smallest natural number such that $\mathsf{uninit}[\texttt{C}]_i$ does not already appear in that assignment. Since no types are ever propagated from `jsr`, no ambiguity can ever arise. See [26] for details.

Given its advantages, this approach to subroutines could replace the one in JS. This impacts on the description of the whole data flow analysis, because sets of type assignments must be used, instead of single type assignments. It also slightly impacts on the description of object initialization, as just mentioned, but in a minor way; the essence of the approach (i.e. the indexing of uninitialized types to accurately track aliases) is unchanged. The optimization of merging assignments that do not have distinct return address types in the same location could be described in a separate section of JS; although irrelevant for pure specification, it is useful for implementation purposes.

### *5.9.6. Can subroutines be eliminated?*

As evidenced by the above discussions, subroutines are a major source of complexity for bytecode verification. Even though the approach described in Section 5.9.5 is quite simple, it impacts on the whole verification algorithm, requiring the use of sets of type assignments. If subroutines did not exist, single type assignments would be sufficient; moreover, the harmful interaction with object initialization described in Section 5.8.3 would not happen.

Subroutines were introduced in Java bytecode to avoid code duplication when compiling `finally` blocks, but it has been found that very little space is actually saved in mundane code [21,27]. It is widely conjectured that it might have been better not to introduce subroutines in the first place.

While future Java compilers could simply avoid the generation of subroutines, future versions of the JVM must be able to accept previously compiled code that may have subroutines. In other words, the need for backward compatibility prevents the total elimination of subroutines.

A possible approach is to in-line subroutines prior to bytecode verification. After in-lining, the simpler data flow analysis with single type assignments can be run. However, in certain cases it is not immediate how to determine subroutine boundaries for in-lining. For instance, the subroutine in Figure 6 can be exited via a branch. In general, some analysis is needed to determine subroutines' boundaries.

The question is whether it is better to in-line subroutines and then run a slightly simpler data flow analysis, or to run directly a slightly more complex data flow analysis. In order to provide a proper answer, formalizations of in-lining strategies are needed, along with experimental measures comparing the two alternatives. To the author's knowledge, there is currently no such work. A formalization of in-lining should also include a proof of equivalence between programs with subroutines and their in-lined versions, as well as a characterization of the programs with subroutines amenable to the formalized in-lining strategy (ideally, all).



## 6. RELATED WORK

There are a large number of publications [5,11–15,17–24,26–43] that clarify key issues in bytecode verification, propose more precise descriptions of (aspects of) it, and expose certain inadequacies in JS. There are also several publications [8,10,44–47] studying dynamic class loading and its close relationship with bytecode verification.

To the author's knowledge, this paper is the only work to provide a comprehensive analysis of the official specification of bytecode verification and comprehensive suggestions for improvement. Some details of the analysis and improvements, such as those concerning subroutines and class loading, are thoroughly covered in other papers which are explicitly referenced by this paper. An early version of this paper is [48].

A complete formalization of bytecode verification that follows the suggestions given in Section 5 is presented in [26]. This formalization, or at least an informal description of it, could become part of JS. Since the formalization is a simple data flow analysis, its description would be similar to the current one.

In 2000, the author derived an implementation of bytecode verification from the formalization in [26] by means of Specware [49], a tool for the formal specification and refinement of software. This bytecode verifier can serve as a high-assurance reference implementation against which other implementations can be tested.

In the Connected, Limited Device Configuration (CLDC) variant of the JVM [50], intended for resource-constrained embedded systems, bytecode verification is split into an off-device and an on-device phase, as proposed in [41]. Essentially, the off-device phase computes a solution to the data flow analysis problem (i.e. a type assignment to the operand stack and local variables for each instruction address) and enhances the class file with enough information for the on-device phase to reconstruct and check the solution with a space-efficient and time-efficient linear scan of the code. This splitting does not change the nature of the types and their manipulation; the only difference is that the solution is computed, recorded, and checked. Thus, the core of the analysis of JS presented in this paper is directly relevant to this approach to bytecode verification.

### ACKNOWLEDGEMENTS

The author gives special thanks to Jim McDonald for contributing to the discovery of the incorrect wording about protected members described in Section 4.6, Gilad Bracha for many useful discussions, and the anonymous referees whose comments have been helpful in improving the accuracy and presentation of the paper.

### REFERENCES

1. Arnold K, Gosling J, Holmes D. *The Java™ Programming Language* (3rd edn). Addison-Wesley: Cambridge, MA, 2000.
2. Gosling J, Joy B, Steele G, Bracha G. *The Java™ Language Specification* (2nd edn). Addison-Wesley: Cambridge, MA, 2000.
3. Lindholm T, Yellin F. *The Java™ Virtual Machine Specification* (2nd edn). Addison-Wesley: Cambridge, MA, 1999.
4. Gong L. *Inside Java™ 2 Platform Security*. Addison-Wesley: Cambridge, MA, 1999.
5. Dean D, Felten E, Wallach D. Java security: From HotJava to Netscape and beyond. *Proceedings IEEE Symposium of Security and Privacy*, 1996; 190–200.


6. Gosling J. Java intermediate bytecode. *Proceedings of the Workshop on Intermediate Representations (IR'95) ACM SIGPLAN Notices* 1995; **30**:111–118.
7. Nielson F, Nielson HR, Hankin C. *Principles of Program Analysis*. Springer: Berlin, 1998.
8. Saraswat V. Java is not type-safe. *Technical Report*, AT&T Research, 1997. http://www.research.att.com/vj/bug.html.
9. Liang S, Bracha G. Dynamic class loading in the Java™ virtual machine. *Proceedings of the 13th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98) ACM SIGPLAN Notices* 1998; **33**:36–44.
10. Qian Z, Goldberg A, Coglio A. A formal specification of Java class loading. *Proceedings of the 15th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'00)* (*ACM SIGPLAN Notices*, vol. 35). ACM Press: New York, 2000; 325–336. http://www.kestrel.edu/java. (Long Version)
11. Coglio A, Goldberg A. Type safety in the JVM: Some problems in Java 2 SDK 1.2 and proposed solutions. *Concurrency and Computation: Practice and Experience* 2001; **13**(13):1153–1171.
12. Tozawa A, Hagiya M. Careful analysis of type spoofing. *Proceedings of Java-Informations-Tage 1999 (JIT'99)*. Springer: Berlin, 1999; 290–296.
13. Goldberg A. A specification of Java loading and bytecode verification. *Proceedings of the 5th ACM Conference on Computer and Communications Security (CCS'98)*, 1998; 49–58.
14. Qian Z. A formal specification of Java™ Virtual Machine instructions for objects, methods and subroutines. *Formal Syntax and Semantics of Java™* (*Lecture Notes in Computer Science*, vol. 1523), Alves-Foss J (ed.). Springer: Berlin, 1999; 271–312.
15. Coglio A. Treatment of protected members in Java bytecode verification. *Technical Report*, Kestrel Institute. http://www.kestrel.edu/java.
16. Bracha G. Private communication, June 2001.
17. Coglio A. Java bytecode verification: Another complete formalization. *Technical Report*, Kestrel Institute. http://www.kestrel.edu/java.
18. Freund S, Mitchell J. A type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems* 1999; **21**(6):1196–1250.
19. Stata R, Abadi M. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems* 1999; **21**(1):90–137.
20. Coglio A. Java bytecode subroutines demystified. *Technical Report*, Kestrel Institute. http://www.kestrel.edu/java.
21. O'Callahan R. A simple, comprehensive type system for Java bytecode subroutines. *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL'99)*, 1999; 70–78.
22. Freund S, Mitchell J. A type system for Java bytecode subroutines and exceptions. *Technical Note STAN-CS-TN-99-91*, Computer Science Department, Stanford University, August 1999.
23. Coglio A. Simple verification technique for complex Java bytecode subroutines. *Proceedings of the 4th ECOOP Workshop on Formal Techniques for Java-like Programs*, June 2002; http://www.kestrel.edu/java. (Long Version)
24. Stärk R, Schmid J. The problem of bytecode verification in current implementations of the JVM. *Technical Report*, Department of Computer Science, ETH Zürich, 2000.
25. The Kimera project Web site. http://kimera.cs.washington.edu.
26. Coglio A. Java bytecode verification: A complete formalization. *Technical Report*, Kestrel Institute. http://www.kestrel.edu/java.
27. Freund S. The costs and benefits of Java bytecode subroutines. *Proceedings of OOPSLA'98 Workshop on Formal Underpinnings of Java*, 1998.
28. Casset L, Lanet JL. A formal specification of the Java bytecode semantics using the B method. *Proceedings of the 1st ECOOP Workshop on Formal Techniques for Java Programs*, June 1999.
29. Coglio A, Goldberg A, Qian Z. Towards a provably-correct implementation of the JVM bytecode verifier. *Proceedings of OOPSLA'98 Workshop on Formal Underpinnings of Java*, October 1998.
30. Fong P, Cameron R. Proof linking: Modular verification of mobile programs in the presence of lazy, dynamic linking. *ACM Transactions on Software Engineering and Methodology* 2000; **9**(4):379–409.
31. Freund S, Mitchell J. A formal framework for the Java bytecode language and verifier. *Proceedings of the 14th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99) ACM SIGPLAN Notices* 1999; **34**:147–166.
32. Hagiya M, Tozawa A. On a new method for dataflow analysis of Java Virtual Machine subroutines. *Proceedings of the 5th Static Analysis Symposium (SAS'98)* (*Lecture Notes in Computer Science*, vol. 1503). Springer: Berlin, 1998; 17–32.
33. Jones M. The functions of Java bytecode. *Proceedings of OOPSLA'98 Workshop on Formal Underpinnings of Java*, October 1998.
34. Klein G, Nipkow T. Verified lightweight bytecode verification. *Proceedings of the 2nd ECOOP Workshop on Formal Techniques for Java Programs*, June 2000.




35. Leroy X. Java bytecode verification: An overview. *Proceedings of the 13th Conference on Computer Aided Verification (CAV'01)* (*Lecture Notes in Computer Science*, vol. 2102). Springer: Berlin, 2001; 265–285.
36. Nipkow T. Verified bytecode verifiers. *Proceedings of the 4th Conference on Foundations of Software Science and Computation Structures (FOSSACS'01)* (*Lecture Notes in Computer Science*, vol. 2030). Springer: Berlin, 2001; 347–363.
37. Posegga J, Vogt H. Java bytecode verification using model checking. *Proceedings of the OOPSLA'98 Workshop on Formal Underpinnings of Java*, October 1998.
38. Pusch C. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. *Proceedings of the 5th Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99)* (*Lecture Notes in Computer Science*, vol. 1579). Springer: Berlin, 1999; 89–103.
39. Qian Z. Standard fixpoint iteration for Java bytecode verification. *ACM Transactions on Programming Languages and Systems* 2000; **22**(4):638–672.
40. Requet A. A B model for ensuring soundness of a large subset of the Java Card virtual machine. *Proceedings of the 5th ERCIM Workshop on Formal Methods for Industrial Critical Systems (FMICS'00)*, April 2000; 29–45.
41. Rose E, Rose K. Lightweight bytecode verification. *Proceedings of the OOPSLA'98 Workshop on Formal Underpinnings of Java*, October 1998.
42. Stärk R, Schmid J, Börger E. *Java™ and the Java™ Virtual Machine: Definition, Verification, Validation*. Springer: Berlin, 2001.
43. Yelland P. A compositional account of the Java Virtual Machine. *Proceedings of the 26th ACM Symposium on Principles of Programming Languages (POPL'99)*, 1999; 57–69.
44. Drossopoulou S. An abstract model of Java dynamic linking and loading. *Proceedings of the 3rd Workshop on Types In Compilation (TIC'00)* (*Lecture Notes in Computer Science*, vol. 2071). Springer: Berlin, 2001; 53–84.
45. Fong P, Cameron R. Proof linking: Distributed verification of Java classfiles in the presence of multiple classloaders. *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium (JVM'01)*. USENIX, 2001; 53–66.
46. Jensen T, Le Métayer D, Thorn T. Security and dynamic class loading in Java: A formalisation. *Proceedings 1998 IEEE International Conference on Computer Languages (ICCL'98)*, 1998; 4–15.
47. Tozawa A, Hagiya M. Formalization and analysis of class loading in Java. *Higher-Order and Symbolic Computation (HOSC)* 2002; **15**:7–55.
48. Coglio A. Improving the official specification of Java bytecode verification. *Proceedings of the 3rd ECOOP Workshop on Formal Techniques for Java Programs*, June 2001.
49. Kestrel Institute and Kestrel Technology LLC. Specware™. http://www.specware.org.
50. Sun Microsystems. Connected, limited device configuration: Specification version 1.0a. http://java.sun.com/j2me.