A Specification of Java Loading and Bytecode Verification

Allen Goldberg
Kestrel Institute
3260 Hillview Av.
Palo Alto, CA 94304
goldberg@kestrel.edu

December 22, 1997

Abstract

This paper gives a mathematical specification the *Java Virtual Machine (JVM)* bytecode verifier. The specification is an axiomatic description of the verifier that makes precise subtle aspects of the *JVM* semantics and the verifier. We focus on the use of data flow analysis to verify type-correctness and the use of typing contexts to insure global type consistency in the context of an arbitrary strategy for dynamic class loading. The specification types interfaces with sufficient accuracy to eliminate run-time type checks. Our approach is to specify a generic dataflow architecture and formalize the *JVM* verifier as an instance of this architecture. The emphasis in this paper is on readability of the specification and mathematical clarity. The specification given is consistent with the descriptions in the Lindholm's and Yellin's *The Java™ Virtual Machine Specification.* It less committed to certain implementation choices than Sun's version 1.1 implementation. In particular, the specification does not commit an implementation to any loading strategy, and detects all type errors as early as possible.

1 Introduction

1.1 The Java Virtual Machine

The Java compiler translates Java class definitions into platform-independent target language known as the Java Virtual Machine. The *Java Virtual Machine (JVM)* is a type-safe, stack-oriented abstract machine. Type safety requires that programs with type errors must produce an error indication rather than execute and produce an erroneous result. Type safety is an important property of a language because it aids in development and debugging, but particularly because the erroneous executions resulting from executing programs with type violations can be exploited to introduce security flaws. *JVM* code (also called bytecode), not Java™ source is transmitted when an "applet" is sent over the Internet and remotely executed. Because the transmitted code cannot be trusted to be the unmodified output of a correct Java (or other language) compiler, the code must be checked for consistency either prior to execution or by run-time checking.

A *JVM* program consists of a collection of class (including interface) definitions that are dynamically loaded into the execution environment. A program is type safe if each class is type consistent within itself and with respect to other classes already loaded into the environment. Thus we formulate the type safety problem as: given a consistent global typing environment and a class file, determine if the class file is well-typed given the current global typing environment. If it is, then extend the global typing environment with the additional declarations and subtyping relationships derived from the class description.

The bytecode verifier performs static checks and dynamic checks. The static checks insure that a class definition can be parsed to yield a constant pool and syntactically-correct code for each method. The constant pool, which is a symbol table, has entries for each method and field defined in the class and for each class, field, and method referenced in the method code. It also identifies a class' direct superclass and the interfaces it directly implements. For class files that define interfaces, it identifies its direct superinterfaces. The static checks performed insure that every literal referenced in the code of a method is described by and is consistent with an entry in the constant pool for the class.

This work is supported by DARPA contract F30602-96-C-0363

The type correctness of a class definition is difficult to verify because the JVM is *weakly typed*. The *JVM* is a stack machine manipulating an operand stack, a set of local variables, or registers, and a heap containing object instances. The type of stack positions and local variables, like hardware registers, vary during method execution. Thus, for example, for a method to be type safe, it must be the case that whenever a floating add instruction `fadd` is executed the top two stack elements must be typed as floating point numbers. This check can be performed during execution, by labeling stack elements with type tags, but this introduces an unacceptable run-time inefficiency. The dynamic checks of the verifier verify the type safety and related properties of all possible execution paths of each method defined in the class. Providing such a proof for all correct programs is an undecidable problem. There is, though, a decidable and efficient method of computing a conservative estimate in which some programs that are type safe are incorrectly rejected, along with all type inconsistent programs. Using a conservative approximation *when it is clear what subset is accepted* is not problematical since then Java compilers can be designed to generate JVM code that falls outside of the subset of type-correct but rejected programs.

In the JVM, when a method is invoked it executes with a new empty stack with actual parameters loaded into local variables; when it returns, either normally or because of an uncaught exception, control is transferred to the calling method. The caller's execution environment (in particular, its own operand stack and local variables) is restored and updated. Although methods may run concurrently, a method cannot access or modify another method's local variables or stack. Thus given a *global typing context* of signature and subtyping information, bytecode verification can be performed independently on each method of the class.

1.2 Dynamic Loading

Unfortunately, the complete global typing context of a class is generally not available when a class is verified. One reason for this is that classes may reference the other recursively. More fundamentally, the *JVM* specification permits a class to be loaded at the latest possible moment—when a method of the class is invoked.

Dean [D97] considers how to insure global type consistency in a dynamically loaded environment. The key condition is *monotonicity* of the global typing context. Intuitively, this means that only consistent additions are made to the typing context as classes are loaded. The model of bytecode verification presented here is insures global type correctness in the following sense. The verifier inputs a class definition and a global typing context; it rejects a class definition that is either internally inconsistent or inconsistent with the global typing context. If it accepts the class definition, it generates an extended global typing context. If the global typing context is sequentially threaded to all invocations of the verifier, global consistency is assured.

The Sun JDK 1.1 implements one strategy for dynamic loading and its bytecode verifier is specialized to this loading strategy. The JDK 1.1, when verifying of a class `c`, it will dynamically load any classes referenced by `c` that are required to insure that all classes are correctly typed. It uses a lazier strategy to check interface types. In this paper, the bytecode verifier we specify is not committed to any particular strategy for dynamic loading nor does it rely on run-time checks to insure type safety.

1.3 Limitations

There are two limitations of this paper. It does not consider enforcement of restrictions specified by access flags, such as `private`, `protected`, `abstract`, and `final` and excludes the `jsr` and `ret` instructions. While treatment of access flags are straightforward, `jsr`/`ret` adds significant complication to the dataflow analysis.

1.4 Outline of this Paper

The next section describes aspects of the JVM type system relevant to this paper. Next, we specify a generic data flow architecture. Then we instantiate the data flow architecture to the *JVM* type analysis problem. We then consider the problem of verifying that object instances have been properly initialized. We then discuss our plans to implement the specified verifier using the Specware system. This is followed by a discussion of related work and some conclusions. The paper assumes familiarity with the *JVM*. The reader is referred to Lindholm's and Yellin's *The Java™ Virtual Machine Specification* [LY97].

2 The JVM Type System

The *JVM* type system is largely derived from the *Java* language, but it also differs from *Java* in significant ways.

2.1 Primitive Types

The complexity surrounding the primitive types derives from low-level efficiency and portability issues. The elements of the stack and local variables of the *JVM* are "words" that hold at least 32 bit. `long` and `double` are held in two consecutive words. On the other hand, in arrays and objects instances values that can be stored with fewer than 32 bits may be packed into bits or bytes.

The types explicated in the *JVM* specification are:

- `byte`, 8 bit signed two's complement integers,
- `short`, 16 bit signed two's complement integers,
- `char`, 16 bit unsigned integers representing unicode characters,
- `int`, 32 bit signed two's complement integers,
- `long`, 64 bit signed two's complement integers,
- `float`, 32 bit IEEE 754 floating point numbers,
- `double`, 64 bit IEEE 754 floating point numbers,
- `returnAddress`, denoting an instruction addresses within the method's code. Since we do not treat `jsr/ret`, this type is not used in the specification.

In addition, we define the following types:

- `lLong`, representing the low-order 32 bits of a `long`,
- `hLong`, representing the high-order 32 bits of a `long`,
- `lDouble`, representing the low-order 32 bits of a `double`,
- `hDouble`, representing the high-order 32 bits of a `double`,
- `Boolean`, representing a single bit quantity,
- `Void`, used in method signatures to denote the return type of a method that does not return a value.

Define *StkPrTy*={`int`, `float`, `lLong`, `hLong`, `lDouble`, `hDouble` }. These are the type designations of primitive types on the stack or in local variables.

Define *SigPrTy*={`byte`, `short`, `char`, `int`, `long`, `float`, `double`}. These are the type designations of primitive types in fields and in method signatures.

Define $ArrPrTy = SigPrTy \cup \{$`boolean`$\}$. These are the primitive types that may appear as elements of arrays.

2.2 Reference Types

There are four kinds of reference types: classes, arrays, interfaces and the type `null`, the type with a single value, the reference `null`. Each object and interface type is uniquely named by its fully qualified name together and the name of its class loader [S97]. For the purposes of this paper, the structure of the name space of classes is not relevant. Thus the collection of class names is denoted by an abstract set N. (Should this paper be extended to check access constraints implied by the `protected` attribute, then the package structure of the name space must be formalized.) The pre-defined class `java.lang.object`, loaded by the system loader, is denoted `object` in the set N. Let N_o denote the set of non-array objects, and N_i the set of interfaces. Then $N=N_o \cup N_i$.

Define $BaseArrTy=ArrPrTy \cup N$. *BaseArrTy* defines the set of non-array types that may be elements of an array. Define the set of array terms *ArrTy* as the smallest set closed under the following rules.

- If t is in *BaseArrTy* then the term *[t]*, read *"array of type t"* is in *ArrTy*, and has dimension *1*.
- If t is in *ArrTy* and has dimension i, then *[t]* in *ArrTy*, and has dimension $i+1$.
- All terms in *ArrTy* have dimension at most 255.

The collection of reference types $RefTy = N \cup ArrT \cup \{\texttt{null}\}$. A *stack type* is an element of the set $StkTy = StkPrTy \cup RefTy$. These are the types designations of stack elements and local variables.

In our specification, four binary subtype relations over N are used. $a \preccurlyeq_o b$ *for* $a, b \in N_o$ has the intended meaning that a directly extends b. $a \preccurlyeq_i b$ for $a, b \in N_i$ has the intended meaning that b is a direct superinterface of a. The assertion $a \preccurlyeq_i \texttt{object}$ has the intended meaning that a does not have direct superinterfaces. Finally, $a \preccurlyeq_{imp} b$ for $a \in N_o$ and $b \in N_i$ has the intended meaning that object type a directly implements interface b. These three relations are called *direct subtype relations*. The fourth relation, the *indirect subtype relation*, $a \preccurlyeq b$ for $a, b \in N$, has the intended meaning that (a,b) is in the reflexive transitive closure $\preccurlyeq_o \cup \preccurlyeq_i \cup \preccurlyeq_{imp}$. For any relation r, let $TC(r)$ denote its reflexive transitive closure. Finally, define the set of *signature types*, $SigTy = N \cup ArrType \cup SigPrTy$.

2.3 Global Typing Contexts and Name Resolution

As the *JVM* executes, it loads class definitions. When a class gets loaded a *constant pool* for that class is constructed. The constant pool specifies if the class is an object class or an interface, the signature of fields and methods of the class, the interfaces the class implements, and its superclass or superinterfaces. It also specifies the fully qualified name, the type of fields, and signature and return type of methods referenced by the class. The former are treated as *assertions* true of the defined class, the latter *assumptions* about other classes that must be verified. When a class is verified, the (internal) consistency of the method code is checked against the type information in the constant pool. When the referenced class is loaded, it is possible check the type consistency of the referenced methods and fields. To perform this verification a *global typing context* of assertions and assumptions is maintained.

Let F denote the abstract set of field names and M, method names. A *field signature assertion* is a term of the form $n.f{:}t$ where $n \in N$ is a reference type name, $f \in F$ is a field name and $t \in SigTy$. A *method signature assertion* is a term of the form $n.f{:}\sigma$ where $n \in N$, $m \in M$ is a method name, and σ is a method signature of the form $t^* \rightarrow r$. Here t^* is a sequence of zero or more elements from *SigTy*, and $r \in SigTy \cup \{\texttt{void}\}$. A *class assertion* is a term of the form *n:class* or *n:interface*. A *signature assertion* is either a field signature assertion, a method signature assertion, or a class assertion.

An *assertion set* is a set containing signature assertions and direct subtype assertions.

When a class c is loaded, an assertion set Γ_c, is extracted from its constant pool and each assertion in Γ_c is added into a global assertion set Γ. The typing assertions in Γ_c are determined as follows.

- If c is an object class, then the assertion *c:class* is in Γ_c. If c is an interface then the assertion *c:interface* is in Γ_c. Class files define either classes or interfaces, not arrays or the null reference type.
- If c is class (other than object) its direct superclass is specified by an entry of the constant pool. If the name is s, then the $c \preccurlyeq_o s$ is in Γ_c. If c is an interface, then no superclass assertions are made.
- If c is an interface, for each of its direct superinterface s the assertion $c \preccurlyeq_i s$ is in Γ_c. If c has no superinterfaces the assertion $c \preccurlyeq_i \texttt{object}$ is in Γ_c.
- If c is an object class, for each interface s that c implements, the assertion $c \preccurlyeq_{imp} s$ is in Γ_c.
- For each field name, f, with type t defined, not inherited in c, the field signature assertion $c.f{:}t$ is in Γ_c.
- For each method m defined, not inherited, in c, with signature and return type σ, the method signature assertion $n.f{:}\sigma$ is in Γ_c.

If Γ is an assertion set define $a \preccurlyeq^*_\Gamma b$ if the pair (a,b) is in the reflexive transitive closure of the union of the subtype assertions in Γ.

An assertion set Γ is *inconsistent* if any of following conditions hold.

1. $a{:}class \in \Gamma$ and $a{:}interface \in \Gamma$.
2. $a \preccurlyeq_o b \in \Gamma$ and either $a{:}interface \in \Gamma$ or $b{:}interface \in \Gamma$.
3. $a \preccurlyeq_i b \in \Gamma$ and $a{:}class \in \Gamma$.
4. $a \preccurlyeq_i b \in \Gamma$, $b \neq \texttt{object}$, and $b{:}class \in \Gamma$.
5. $a \preccurlyeq_{imp} b \in \Gamma$ and either $a{:}interface \in \Gamma$ or $b{:}class \in \Gamma$.
6. $a \preccurlyeq_o b \in \Gamma$, $a \preccurlyeq_o b' \in \Gamma$ and $b \neq b'$.
7. There is an n such that $\texttt{object} \preccurlyeq_o n$ or $\texttt{object} \preccurlyeq_i n$ or $\texttt{object} \preccurlyeq_{imp} n$.
8. $\preccurlyeq^*_\Gamma$ contains a non-trivial directed cycle.
9. $a \preccurlyeq^*_\Gamma b$, $a.f{:}t^* \rightarrow r \in \Gamma$, $b.f{:}t^* \rightarrow r' \in \Gamma$ and $r \neq r'$.
10. Suppose $a_1 \preccurlyeq_o a_2 \in \Gamma$, $a_2 \preccurlyeq_o a_3 \in \Gamma, \ldots, a_n \preccurlyeq_o \texttt{object} \in \Gamma$, $a_1 \preccurlyeq_{imp} b \in \Gamma$, $b \preccurlyeq_\Gamma b'$, and $b'.m.\sigma \in \Gamma$, but there is no i such that $a_i.m.\sigma \in \Gamma$. In words, Γ is inconsistent if a class a_1 implements an interface b, all of the superclasses of a_1 have been loaded, and some method that is defined or inherited in interface b_1 is not defined or inherited by a_1.

An *assumption set* is a set containing signature assertions and assertions of the form $a \preccurlyeq b$. When a class c is loaded, an assumption set A_c, is extracted from its constant pool. Each assertion in A_c is added into a global assumption set A. The assertions in A_c are determined as follows.

- If in some method of c a field f, defined in class d, is referenced, then the constant pool has an entry with the name of the defining class, field name and its type t. Then the field signature assertion $c.f{:}t$ is in A_c.
- If in some method in c a method defined in class d is referenced, then the constant pool has an entry with the name of the defining class, field name and signature σ. Then the method signature assertion $c.f{:}\sigma$ is in A_c.
- Furthermore, as methods of c are verified assertions of the form $a \preccurlyeq b$ are added to the assumption set.

A name $n \in N$ is *closed* with respect to an assertion set Γ if the following conditions are met.

- If n is $\texttt{object}$, then n is closed.
- For $n \in N_o$ other than $\texttt{object}$, n is closed if
    - there is exactly one b such that the assertion $n \preccurlyeq_o b \in \Gamma$, and b is closed, and
    - for each $b \in N_i$ such that $a \preccurlyeq_{imp} b \in \Gamma$, b is closed.
- For $n \in N_i$, n is closed if
    - $n{:}interface \in \Gamma$,
    - and for each $b \in N_i$ such that $n \preccurlyeq_i b$, b is closed.

Let A_s be the set signature assertions in A and $A_\preccurlyeq$ the subtypes assertions in A If Γ is an assertion set and A an assumption set, the pair (Γ, A) is *inconsistent* if any of the following are true.

1. Γ is an inconsistent assertion set.
2. $c.m.\sigma \in A$ and either $c{:}class \in \Gamma$ or $c{:}interface \in \Gamma$ but $c.m.\sigma \notin \Gamma$. (A field or method m is asserted to be declared in the class c. The class c has been introduced to the typing context as evidenced by the assertion $c{:}class \in \Gamma$. However, the method m is not declared or has incorrect signature or return type.)
3. $\Gamma \cup A_s$ is an inconsistent assertion set. (Note because of condition (10), an inconsistency assertion set is may become consistent with the addition of signature assertions. Thus $\Gamma \cup A_s$ consistent doesn't imply Γ consistent.)

4. The reflexive transitive closure of $A_{\preccurlyeq} \cup \preccurlyeq^*_\Gamma$ contains a non-trivial directed cycle.
5. $a \preccurlyeq b \in A$, $a{:}interface \in \Gamma$, a is closed with respect to Γ but $(a,b) \notin \preccurlyeq^*_\Gamma$
6. $a_1 \preccurlyeq b \in A$, $a_1{:}class \in \Gamma, b{:}class \in \Gamma$, $a_1 \preccurlyeq_o a_2 \in \Gamma$, $a_2 \preccurlyeq_o a_3 \in \Gamma, \ldots, a_n \preccurlyeq_o \texttt{object} \in \Gamma$ but $b \neq a_i$ for some $i=1,..,n$.
7. $a \preccurlyeq b \in A$, $a{:}class \in \Gamma$, $b{:}interface \in \Gamma$, a is closed with respect to Γ but $(a,b) \notin \preccurlyeq^*_\Gamma$

A consistent pair (Γ, A) is a *typing context*.

In Sun's JDK 1.1.4 implementation, when an interface class gets resolved, all of its superinterfaces are loaded, if necessary, and resolved. In other words, before an interface method can execute, its interface class must be closed. This specification is consistent with a lazier strategy in which may allow a interface to be loaded, and initialized without loading a superinterface, as long as there are no references in the already loaded classes to methods or static fields defined in the superinterface. With this lazy loading strategy it may be that a class that purports to implement an interface but does not, because it fails to implement a method defined a superclass of the interface. If the superinterfcae is subsequently loaded a type error will be reported.

3 A Data Flow Analysis Architecture

Data flow analysis is a methodology used to establish assertions at program points that are invariant over all program executions. A typical assertion that may be computed by flow analysis is "the value of variable *r* at program point *p* is a constant." Assertions are represented by elements of a meet semi-lattice. Intuitively, the meet operation of two lattice points *a* and *b* represents what can be maximally asserted if along one control flow path *a* is true, and along another *b* is true. For each statement in the program, a transfer function is defined that maps an element of the lattice representing an assertion true prior to execution of the statement to a lattice point representing an assertion true after execution of the statement. The basic idea of flow analysis is to symbolically execute via the transfer function the program over the lattice structure using the meet operation to merge the properties true about different execution paths to the same program point. This is computation is a fixed-point computation over the lattice structure that terminates when stability is reached. Under suitable conditions, the least fixed-point solution is characterized as the meet-over-all-paths solution. This means the assertions associated with a program point at the termination of the algorithm are sharpest invariants true of every execution sequence that reaches that point. To instantiate the data flow architecture to a problem, the control flow graph, the lattice and the transfer functions are specified. This paper is organized around the specification of these parameters.

A *meet semi-lattice* is a tuple $L = (U, \sqsubseteq, \sqcap, \perp)$ where U is a universe of elements, $\sqsubseteq$ a partial order on U, $\sqcap{:}U \times U \to U$, is called the *meet operation*, and $\perp \in U$ is the bottom element satisfying the axioms:

- $\sqsubseteq$ is reflexive, anti-symmetric and transitive
- $\perp \sqsubseteq x$ for each $x \in U$,
- $x \sqsubseteq \top$ for each $x \in U$,
- for each x and y, $x \sqcap y \sqsubseteq x$, $x \sqcap y \sqsubseteq y$, and for all a such that $a \sqsubseteq x$ and $a \sqsubseteq y$, $a \sqsubseteq x \sqcap y$.

A *descending chain* is sequence of strictly decreasing elements of U. If all descending chains in a lattice are finite then the lattice satisfies the *descending chain condition.* A function $f{:}U \times U \to U$ is *distributive* if for all a, b in U, $f(a \sqcap b) = f(a) \sqcap f(b)$.

A *data flow problem* consists of a

- a directed graph, called the control flow graph, $G=(V, E)$ with a distinguished entry vertex, *init*,
- a meet semi-lattice, $L = (U, \sqsubseteq, \sqcap)$ with a maximal element, $\top$, satisfying the descending chain condition;
- for each edge e of G a function $TF_e{:}U \to U$ that is distributive over L, and
- an initial value $i \in U$ initializing the data flow at node *init*.

Let $P(v)$ be the set of paths from the initial vertex to v. The *meet-over-all-paths* solution to a dataflow problem is a map $M: V \rightarrow L$ so that $M(v) = \sqcap_{p \in P(v)} TF_p(i)$ where TF is extended from edges to paths by composition.

Theorem. The following algorithm converges and computes the meet-over-all paths to the dataflow problem:

> O = λv.`if` $v = init$ `then` i `else` $\top$;
>
> `while` *there exist an edge* $e=(v, w)$ *such that* $TF_e(O(v)) \sqsubseteq O(w)$ `do`
>
> $\quad O(w) \leftarrow O(w) \sqcap TF_e(O(v))$;
>
> return O;

Proof. See [Mu97] and [K73].

The algorithm can, of course, be refined into efficient implementations.

3.1 Product and Stack Lattices

The *product lattice* $L = L_1 \times \cdots \times L_n$ of meet semi-lattices $L_i = \langle U_i, \sqsubseteq_i, \sqcap_i, \perp_i \rangle$ is a meet semi-lattice whose universe is $U_1 \times \cdots \times U_n$. The partial order and meet operations are defined componentwise, e.g. $<a_1, \cdots, a_n> \sqsubseteq <b_1, \cdots, b_n>$ iff $a_i \sqsubseteq_i b_i$ for each i. $\perp = <\perp_1, \cdots, \perp_n>$

The *coalesced product lattice* $L = L_1 \times \cdots \times L_n$ of meet semi-lattices $L_i = \langle\langle U_i, \sqsubseteq_i, \sqcap_i, \perp_i \rangle\rangle$ is a meet semi-lattice whose universe is $\{\perp\} \cup U_1 - \{\perp_1\} \times \cdots \times U_n - \{\perp_n\}$. The partial order is defined componentwise, $<a_1, \cdots, a_n> \sqsubseteq <b_1, \cdots, b_n>$ iff $a_i \sqsubseteq_i b_i$ for each i. If for any i, $a_i \sqcap_i b_i = \perp_i$, $<a_1, \cdots, a_n> \sqcap <b_1, \cdots, b_n> = \perp$. On other values the meet operation is defined componentwise.

Define $select: L \times i:\{1..n\} \rightarrow L_i$ to be the projection function, $select(<a_1, \cdots, a_n>, i) = a_i$. Define $update: L \times i:\{1..n\}, L_i \rightarrow L$ to update the i^{th} component, $update(<a_1, \cdots, a_n>, i, b) = <a_1, \cdots, a_{i-1}, b, a_{i+1}, \cdots, a_n>$.

Fact 1. *Select* is distributive in its first argument, *update* is distributive in its first and third arguments.

Suppose $L = (U, \sqsubseteq, \sqcap, \perp)$ is a meet-semi lattice. Consider stacks of maximum size s, $s>0$ whose elements are taken from U. Such a stack is itself a meet semi-lattice, denoted $Stack(L, s)$. The universe of $Stack(L, s)$ are all stacks of size up to s with $\perp$ adjoined,; Define $s \sqsubseteq t$ *iff* $size(s)=size(t)$ & $top(s) \sqsubseteq top(t)$ and, inductively, $pop(s) \sqsubseteq pop(t)$. The meet operation is defined similarly. *top* and *pop* are the usual stack operations extended so that *top, push and pop* yield $\perp$ (in the appropriate lattice) if any of its arguments are $\perp$. Also, define $top(empty)=\perp$, $pop(empty)=\perp$, and $push(x, t)=\perp$, if $size(t)=s$.

Fact 2. *pop* is distributive, i.e. $pop(s \sqcap t) = pop(s) \sqcap pop(t)$. Also, *push* is distributive in both arguments.

Fact 3. The composition of distributive functions is distributive.

Note that the type *Boolean* is a meet semi-lattice with *false* as $\perp$, $false \sqsubseteq true$, and $x \sqcap y = x \& y$.

Fact 4. if L is a lattice with a top element $\top$, and $f: L \rightarrow L$ and $p: L \rightarrow Boolean$ are distributive and then so is

$$g(x) = \begin{cases} \top, & \text{if } x = \top, \\ f(x), & \text{if } p(x), \\ \perp, & \text{otherwise.} \end{cases}$$

4 Instantiation of the Dataflow Architecture to the JVM

4.1 Control Flow Graph

The control flow graph, $G=(V, E)$, for a *JVM* method has a vertex for each instruction and edges denoting control flows between instructions. With the exception of the pair of instructions `jsr` and `ret` construction of the flow graph from the class file is straightforward and not formally specified in this paper. This paper does not treat `jsr/ret`, instructions that we believe are best formalized using inter-procedural dataflow methods. We assume the existence of a function that produces a control flow graph from a method in a class file. Such a function must perform the static checks to insure that the code and its exception table are well-formed, must process wide instructions, and must determine control flow edges corresponding to caught exceptions. It is not necessary to explicitly model exceptions that are not caught within the method since these simply terminate execution of the method. An instruction's affect on the stack and on local variables depends on whether an exception is raised or not. Thus, transfer functions are associated with edges, not with vertices. Edges from a statement to exception-handling code are called *exception edges*, non-exception edges are called *normal edges.*

4.2 The Lattice for the JVM

In this section, we define a lattice, L, used for the dataflow analysis of JVM programs. $L=\langle\langle L_g,L_t\rangle\rangle$ is the coalesced product of two lattices, with $\top$ adjoined as a top element. L_g representing the global typing context and L_t the types of the local variables and stack elements.

The universe of L_g are consistent global typing contexts together with $\perp$, which denotes the inconsistent global typing context. Given two consistent typing contexts (Γ_1, A_1) and (Γ_2, A_2), define $(\Gamma_1, A_1)\sqsubseteq(\Gamma_2, A_2)$ if $\Gamma_2\subseteq\Gamma_1$ and $A_2\subseteq A_1$. Define $(\Gamma_1, A_1)\sqcap(\Gamma_2, A_2)$ to be $(\Gamma_1\cup\Gamma_2, A_1\cup A_2)$ if $(\Gamma_1\cup\Gamma_2, A_1\cup A_2)$ is consistent and $\perp$ otherwise.

The lattice L_t specifies information about the type and initialization status of each stack position and local variable of a method. $L_t=\langle\langle L_{stk}, L_{var}\rangle\rangle$ is a coalesced product lattice of the lattice L_{stk} representing the operand stack, and L_{var} representing the local variables. $L_{stk}=Stack(L_e, s)$ where s is the maximum stack size for the method which is given in the class definition. L_{var} is the n-fold product (not coalesced) lattice of L_e, where n is the number of local variables used by the method, which is also given in the class definition.

JVM semantics is such that if a local variable has an inconsistent typing then verification fails only if the variable is used. On the other hand if a stack element has an inconsistent typing then verification fails, regardless of whether the stack value is referenced. This difference is reflected by defining L_{stk} is a coalesced product and L_{var} as a product lattice.

L_e is the lattice used to represent the type and initialization status of individual local variables and stack positions. To motivate the construction of L_e, consider the problem of typing a local variable at a program point. Suppose following one execution path to that point the variable contains a reference to class c_1, and following some other path a reference to class c_2. For the method to be well typed, any subsequent use of that variable must treat the variable as having a type c that widens both c_1 and c_2, and is the most specific type that does so. That is, c should be the least common supertype of c_1 and c_2. This suggests that the class hierarchy be reflected within L_e. However, there are two difficulties with this. First, because classes are loaded dynamically, when the method is verified the least common supertype of c_1 and c_2 may not be known. Second, if either one or both of c_1 and c_2 are interfaces, there is no least common supertype. The alternative is to type variables and stack positions as consistent *sets* of compile-time types with the interpretation that the variable may hold a value consistent with any type in the set. Suppose, for example, a stack position is typed as $\{s_1, \ldots, s_k\}$ for $s_i\in N$. If that stack position is used in the context where a value of type $t\in N$ is required then the assertions $s_i\preccurlyeq t\ \ i=1,\ldots, k$ are added to the typing assumptions.

A subset of *StkTy* is *inconsistent* if it contains two distinct terms such that

- one of the type terms is a primitive type (a consistent set with a primitive type term must be a singleton);
- one is *[t]* and the any non-array type other than `object` or ;
- one term is *[t]* the other *[s]* and $\{s, t\}$ is inconsistent.

The elements of the universe of L_e are consistent subsets of *StkTy*. $\perp$ is adjoined and denotes an inconsistent typing. For *s*, *t* in the universe of L_e different from $\perp$, define $s \sqsubseteq t$ iff $t \subseteq s$ and $s \sqcap t$ as $s \cup t$ if $s \cup t$ is consistent and $\perp$ otherwise.

Since only a finite number of classes are ever loaded into the *JVM*, *L* satisfies the descending chain condition.

4.3 Dataflow Initialization

We assume `object` is the only pre-loaded class. Thus, when the *JVM* is started up the global typing environment $(\Gamma_G, A_G)=(\{\texttt{object}:class\}, \emptyset)$. With respect to verification, loading a class *c* requires updating (Γ_G, A_G) to $(\Gamma_G, A_G) \sqcap (\Gamma_c, A_c)$. If $(\Gamma_G, A_G)=\perp$, the class *c* is rejected and the typing context restored. Then for each method *m* of *c* the dataflow analysis is executed. The dataflow analysis is initialized by assigning a lattice value to the vertex *init* in the control flow graph. The lattice value is determined by the global typing context, and the signature of the method. Recall that when a method is invoked its parameters are placed in local variables, starting at variable 0. The initial value *l* is defined as $l=\langle\langle l_g,l_t\rangle\rangle$, where $l_g=(\Gamma_G, A_G)$, $l_t=\langle\langle l_{stk}, l_{var}\rangle\rangle$. $l_{stk}=\texttt{empty}$, $l_{var}=\langle\{t_1\}, \ldots, \{t_k\}, \perp, \ldots, \perp\rangle$. The types $t_1,\ldots, t_k$ are the types of the parameters of *m*. Note t_1 is the type of `this` if *m* is not a static method. Also, the types `long` and `double` are replaced by `hLong`, `lLong` and `hDouble`, `lDouble` respectively. The data flow algorithm is executed. A verification failure is indicated if the lattice value $\perp$ is assigned to any vertex in the control flow graph. Otherwise, the dataflow algorithm associates an element l_n of the lattice *L* with each vertex of the control flow graph. From this result, an updated global typing context is obtained by taking the meet of the global typing context at each vertex. i.e. $(\Gamma_G, A_G)=\sqcap_{n\ in\ V}\ project(l_n,1)$. If (Γ_G, A_G) is updated to be $\perp$ verification fails. Otherwise (Γ_G, A_G) is used to initialize the dataflow analysis for the next method of the class *c*. This procedure is iterated until all the methods are analyzed. The final global typing context is then passed to the next class that is loaded.

To adhere to the dataflow framework, the global typing context is part of the lattice, and so there is an instance of the global typing context stored for each node of the control flow graph. In an actual implementation, it is only necessary to maintain a single global typing context.

4.4 Transfer Functions

In the dataflow framework, the transfer functions formalize the typing rules for each *JVM* instruction. The bytecode verifier constructs a transfer function for each instruction from the instruction's opcode and operand, and from the constant pool. In this section we define a few generic distributive functions that are useful building blocks for constructing transfer functions.

The transfer function associated with the edge (v, w) expresses a typing rule for the instruction at vertex *v*. The typing rule for exception edges differs from the one for normal edges. If an exception is raised, *JVM* semantics dictate that all the values on the stack are popped, and that the object representing the exception is pushed onto the stack. The local variables are untouched. The transfer function for exception edges need also to make some additional checks as described in section5.

As with typing rules, the transfer functions specify *enabling conditions* on the typing environment, i.e. on the lattice value associated with the vertex. If the condition is not satisfied, the transfer function yields $\perp$, indicating that the method is not well typed. If the condition is satisfied, then the lattice value is transformed according to the semantics of the instruction. Thus, transfer functions are generally of the form of the function *g* in Fact 4 of section 3.1. Since the global typing context will not contain complete information about subtype relations on classes, the expected enabling conditions cannot be checked. Instead, the transfer function adds subtyping assertions to the assumption set of the typing context.

Transfer functions may be specified by describing their behavior on the sub-lattices of *L*, For the stack lattice, the enabling condition tests whether the top *k* elements of the stack exist and satisfy type constraints derived from the instruction. Define *TypMatch=StkPrTy* $\cup${`object`, `single`, `doublel`, `doubleh` [`object`])$\cup${[t]| $t \in$ *StkPrTy*}. The newly introduced "types" `single`, `doublel`, `doubleh` are used in stack manipulation instructions such as dup are only concerned with whether a type is represented in a single or double word. `single` will match (i.e. is $\preccurlyeq$) any single word type such as `int` or any *RefTy;* `doublel` will match `lDouble` and `lLong`, and `doubleh` will match `hDouble` and `hLong`. Recall, *Stack*(*TypMatch, s)* are stacks of size *s* (the maximum stack size for the method) whose elements are from *TypMatch.* Let *chkStk:Stack*(*TypMatch*), $L_{stk} \rightarrow$ *Boolean.*

$chkStk\ (p, l) \Leftrightarrow p = empty \vee size(p) \leq size(l)\ \&(\ \forall x \in top(l))\ top(m) \preccurlyeq\ x)\ \&\ chkStk(pop(p), pop(l)).$

Note `object` $\preccurlyeq t$ is *true* iff $t \in RefType$. It is not difficult to prove that *chkStk* is distributive in its second argument. Consider, for example, the `aastore` instruction; it requires the top of the stack to be typed as *s*, *int*, *[t]* where *t* and *s* are reference types and $s \preccurlyeq t$. The call to *chkStk* with first argument <`object`, *int*, [`object`]> checks the appropriate enabling condition for the `aastore` instruction. The transfer function updates the L_{stk} lattice by popping its three top elements off the stack, and updates the global typing environment with the addition of subtyping assertions to the assumption set. In this case the typing for the top stack element is a set that contains reference types, $\{s_1,\ldots, s_n\}$ and the third argument is a set containing arrays of references $\{[t_1], \ldots, [t_m]\}$. The subtype assertions that must be added to the global typing assumption set are those in the set $\{norm(s_i \preccurlyeq t_j)|\ s_i \in \{s_1,\ldots, s_n\}\ ,\ t_j \in \{t_1,\ldots, t_m\}\}$. A term in *SigTy* may denote an array type, but global typing contexts do not contain such terms. Thus, if $s, t \in SigTy$, and $s \preccurlyeq t$ is to be asserted, then $s \preccurlyeq t$ is first normalized, via the function *norm,* to a subtype assertions in *N*. For example, normalization reduces `null`$\preccurlyeq s$ to *true*, `object`$\preccurlyeq[a]$ to *true,* $[[a]] \preccurlyeq [[b]]$ to $a \preccurlyeq b$, and $[a] \preccurlyeq [[b]]$ to *false.*

Assuming *chkStk* has verified the type stack, the transfer function to update the stack, are composed from *push, pop, top* perhaps binding popped values to variables, so they can be pushed back on the stack. One other function, *element-type*, which maps a set of array types to a set of the corresponding element types, is also needed. We give three examples of transfer functions that update the stack representation. Nearly all transfer functions can be defined as simple compositions of the functions defined.

Instruction	***chkstk* pattern**	**Transfer function**
`fadd`	`int`, `int`	`λx.pop(x)`

`aaload`	`int`, *[t]*	`λx.let`
		`    e-t= element-type(top(pop(x))`
		`in`
		`    push(e-t, pop(pop(x)))`

`dup_x1`	`single`, `single`	`λx.let`
		`   t1=top(l),`
		`   t2=top(pop(l))`
		`in`
		`    push(t1, push(t2, push(t1, pop(pop(l))))`

4.4.1 Instructions on Primitive Types

Many *JVM* instructions, including all of the arithmetic operations, manipulate primitive types on the stack. The transfer functions for these instructions are the identity on global type, and local variable lattices, and modify the stack using the methods described above.

Load and store instructions for primitive values are also straightforward, using methods to verify and update local variables that are analogous to the methods described for stacks. One subtlety is that instructions that write into a local variable that holds one word of a `double` or `long`, must also update the other word to $\perp$.

4.4.2 Object Creation and Manipulation Instructions

4.4.2.1 Array Instructions

The transfer function for `aaload` was given above. The transfer function for the `anewarray` instruction pushes the type of the array onto the stack. The type is determined by lookup into the constant pool. The `baload` instruction will pop either [`boolean`] or [`byte`] off the top of the stack and replace it with an `int`. Other array operations are straightforward.

4.4.2.2 Non-array Instructions

The `new` instruction creates a new object instance. The next section describes how flow analysis is used to track that newly-created objects are properly initialized. Transfer functions for field access instructions, such as `getfield`, are easily constructed using data in the constant pool. These transfer functions add subtype assertions to the global assumption set.

Of the four method invocation instructions, the `invoke_special` instruction is the most complex. This instruction is used for invoking instance initialization methods, private methods, or a method of a superclass of the current class. The typing of this instruction is dependent on which case arises. Case discrimination is statically determined by its operand. The operand of the instruction indexes the constant pool and retrieves a class, c, method, m, and signature, σ, If the method name is <init> then the instruction is used for method initialization. Typing of instance initialization methods is discussed in the next section. If the method has the `private` access flag, then this is an invocation of a private method. Otherwise, it is the invocation of a superclass method. Although this paper has not treated access flags, this context is the most complex situation where they are used and so we consider them here.

In each case, the top of the stack should contain an object reference followed by the parameters. Type checking the parameters is the same in each case. The type constraints on the methods's pararameters are derived from the signature σ and are enforced using the techniques describe above. The difference is varying subtyping requirements on the current class, *cc*, (i.e. the class that is currently being verified, the class *c*, and the object reference on the top of the stack. For an <init> invocation, the type of the object reference must be an uninitialized object. For invocation of a superclass method, the subtype assertion $cc \preccurlyeq c$ is added to the global type assumptions. If the method is `private`, the assertions $cc \preccurlyeq c$ and $c \preccurlyeq cc$ are added. For superclass and private invocations, for each reference type *t* the set of types for top of the stack, $t \preccurlyeq c$ is added to the global type assumptions.

5 Object Initialization

A further objective of the bytecode verifier is to insure that accepted programs do not use an object instance unless it has been properly initialized. In the Java programming language, invoking a constructor method allocates memory for a new class instance and initializes its fields according to user-specified code. In the *JVM*, allocating memory for a new class instance is achieved by the `new` instruction. The new object's fields are initialized by executing a method called <init> compiled from a Java constructor for the class. The bytecode verifier assures that objects that have been allocated with `new` but not yet initialized by invoking <init> cannot be used.

The `new` instruction initializes the fields of the object with default values for each type. Thus, type safety is assured, even if <init> is not called. Nonetheless, the security of the *JVM* is dependent upon executing a proper initialization sequence, since user-defined classes such as extensions to the `classLoader`, must meet security-critical interface requirements, that are at least partially satisfied by insuring proper initialization.

The lifecycle of object creation is follows. First, in some method *m*, a new instance of class *c* is created by execution of a new instruction with the name *c* as its operand. The instruction places a reference to the newly-created-but-not-yet-initialized object on the top of the stack. The reference can be stored in local variable, duplicated on the stack, but may not have its fields referenced or updated, its method invoked, be passed as an argument in a method call, assigned as the value of a field of some other object, or be otherwise "used" until an <init> method of class *c* is called with the reference as the this parameter. Finally, the called <init> method must itself call an <init> method of *c*'s superclass (assuming *c* is not the class object), or another <init> method for *c* with different signature, before it may use the object or return normally.

A method may invoke the same new instructions many times, or there may be many new instructions in the code, so the task of pairing calls to <init> with executions of new is, in general, intractable. To make the problem tractable, the *JVM* rejects programs that have two simultaneous uninitialized instances allocated by the *same textual occurrence* of the new instruction.

Dataflow methods are well-suited to performing these checks. Define a new type term *uninit(i, c)* where *i* is the index of a new instruction within a given method, and *c* is the name of the class allocated by that instruction. In addition, we define *needs-super* as a new type term. Both *uninit(i, c)* and *needs-super* are in *StkTy*. The definition of an inconsistent subset of *StkTy* is extended so that any non-singleton set containing *uninit(i, c)* or *needs-super* is inconsistent.

In our formulation, the enabling condition of the typing rule for the new *c* instruction at location *i* requires that there are no instances of *{uninit(i, c)}* on the stack or in local variables. This formulation is not in strict adherence to *JVM* semantics. *JVM* semantics requires that when a backward branch is executed there are no instances of *{uninit(i, c)}* on the stack or in local variables. Our rule seems simpler and more to the point. The transfer function for new pushes *{uninit(i, c)}* onto the stack representation.

The enabling condition of the typing rule for the subcase of the invoke_special instruction, used to invoke an <init> method from class *c* requires *uninit(i, c)* or *needs-super* as the type of its this argument. If this is *uninit(i,c),* then the method must be an <init> method of class *c.* If this is *needs-super* then the method must be an <init> method of the current class or its direct superclass. The dataflow analysis of the called <init> method is initialized so that the type of local variable zero, which receives the this argument is *needs-super.* If *c* is object, and so has no superclass, the type of local variable zero is initialized to object.

The transfer function for the <init> method subcase of the invoke_special instruction pushes *{c}* onto the representation of the stack. If the this argument is *uninit(i, c)* then all occurrences of {*uninit(i, c)}* on the stack and in local variables is updated to *{c}.* If the "this " argument is *needs-super* then all occurrences of *needs-super* are updated to *{c}.* The typing rules are summarized in the table below.

<table>
<tr><th>Instruction</th><th>Typing Conditions</th><th colspan="2">Transfer Function</th></tr>
<tr><td><code>new</code> c
at location i</td><td>There is no occurrences of uninit(i,c) on the representation of the stack or local variables.</td><td colspan="2">Push {uninit(i,c)} onto the representation of the stack.</td></tr>
<tr><td><code>invoke_special</code>
invoking an <code><<i>init</i>></code> method</td><td>• The argument list must be correctly typed.
• The <code>this</code> argument must have type <i>uninit(i,c)</i> or <i>needs-super</i>.
• If <code>this</code> is <i>uninit(i,c)</i>, then the method must be an <code><init></code> method of class <i>c</i>.
• If <code>this</code> is <i>needs-super</i> then the method must be an <code><init></code> method of the current class or its direct superclass.</td><td><code>c.<init></code> initialized so that local variable 0 has type <i>needs-super</i>. Other local variables hold the types of the argument list.</td><td>Upon return from <code><init></code>, <i>{c}</i> is pushed on stack and each occurrence of {<i>uninit(i,c)}</i> or {<i>needs-super}</i> is updated to <i>{c}</i></td></tr>
<tr><td><code>return</code></td><td>There are no occurrences of needs-super.</td><td colspan="2"></td></tr>
</table>

In addition, *JVM* semantics require that there must never be an uninitialized class instance in a local variable protected by an exception handler or a finally clause. “finally” is a Java construct that compiled in the JVM as a “subroutine.” This requires that the transfer function on any exception edge imposes the condition that there are no occurrences of *uninit(i,c)* or *needs-super* in the sets representing the types of local variables. It also imposes the same condition on certain edges exiting `jsr` instructions, but that is beyond the scope of this paper.

6 Specware

This paper has given in informal mathematical notation, a reasonably precise formalization to the core functionality of the bytecode verifier. It has only loosely described functions that extract from a class file a control flow graph, and transfer functions. We plan to specify all of this using the Specware system available from Kestrel Institute.

Specware[SJ95] supports the formal development of programs from specifications. In Specware, basic specifications are theories in high-order logic. Complex specifications are composed from basic specifications using high-level module operations that include parameterization. Thus constructions such as instantiating the generic data flow architecture to the *JVM*, and constructing product and stack lattices from other lattices are nicely expressed in Specware.

The unit of refinement is the *interpretation*, a theorem-preserving translation of the vocabulary of a source specification into the terms of a target specification. Specware makes available a theorem prover to prove interpretations correct and to prove putative properties of the specification. Specware supports the generation of code in Lisp and C++.

Thus, using Specware, provably-correct code can be generated from our specification. The required theories and proofs are currently under development. Implementing the specification is largely a matter on selecting data structures and refining the dataflow algorithm into more efficient forms, for example by maintaining a workset of the vertices that require updating [CP88].

7 Related Work

The application of dataflow analysis to type inference is an old idea, used in SETL, a weakly-typed, high level language with sets, maps, sequences, etc. as data types [Te74].

Most closely related to our work is the work of Qian [Q97] who is also formalizing JVM semantics and the behavior of the bytecode verifier. We believe our formulation is crisper; for example, ours makes it clear how type information from different control flow paths is merged and the requirement of distributivity of transfer functions. We treat arrays and all primitive types, and are explicit about stack overflow. He treats the `jsr/ret` instructions. The dJVM [C97] defines an interpreter for the *JVM* using ACL2, a functional language with an associated proof system. The dJVM insures type safety at runtime using type tags and so does not yet address the bytecode verifier.

The English, official *JVM* specification by Linholm and Yellin is quite precise and well organized. We found an obfuscation on page 130 where the merging of the typing of the stack and local variables is described. "If both local variables contain a reference, then the merged state contains a reference to the first common superclass of the two types." The statement is technically correct assuming that "class" excludes interfaces. For interfaces, the first common *superclass* of the two interfaces i_1 and i_2 is `java.language.object`. This is in fact what the code does, but then should a value typed as `object` be used in a context where a common superinterface of i_1 and i_2 is required, type checking should fail. Of course, it doesn't. The verifier lets this case through and run-time checks are used to insure type safety. Both this paper and Qian [Q97] recognized that if the bytecode verifier uses sets of types to characterize the possible types of local variables, then runtime checks can be avoided. However, there is not much gained in doing so because invoking an interface method requires a search of the method table of the `this` pointer's object class. The type test corresponds to searching the table but not finding a name/signature match.

Saraswat in his paper "Java is not type safe" [S97] describes a bug in the JVM due to class name spoofing. It suggests that a formal specification of namespaces management and loading, particularly in a muti-threaded environment, should be pursued. Dean initiated such a study in [D97].

The Kimera project [K97] has uncovered bugs in the JVM using mutation analysis. They have written their own bytecode verifier. They take JVM programs mutate them and run both verifiers. If they get different results then a potential bug site has been exposed. This testing approach nicely complements formal method approaches.

Nipkow in his paper "Java-light is type safe — definitely" [N97] presents a formalization of the Java type system, an operational semantics for a significant subset of Java, and a proof of type soundness using Isabelle/HOL. We have not considered an operational semantics for the JVM and have not proved a type safety result. However since the type system of Java and the JVM are closely related, his rules characterizing a well-formed typing environment closely correspond to our definition of a consistent global typing context.

8 Conclusions

We claim our specification is clear and explicit about key issues in the semantics of the JVM. At the same time, the specification is directly implementable by either manual or automated methods. Furthermore our specification is not committed to a loading strategy and does not require run-time checks on interface types.

The use of the bytecode verifier to establish that object instances are properly initialized illustrates the flexibility of dataflow analysis. We believe that there are other analysis tasks specific to Java that require dataflow analysis. These include:

- Program optimizations that reduce the number of array bound checks. or null de-referencing.
- Constraints on class loaders. A significant feature of Java is that it Java permits user-defined class loaders. However, this has lead to bugs because these loaders did not satisfy interface requirements. These interface requirements can be verified by an extended bytecode verifier.
- Finer type analysis for security or other applications. Type systems are a good vehicle to specify security models. Dataflow analysis is an effective mechanism to statically verify conformance to these models. [V97]

Thus, it is desirable to design a bytecode verifier that permits extension. Our specification and the code that derives from it have the necessary modularity and locality to support such extensions. By making "monotonic" additions or refinements to the lattice, the safety guarantees of the verifier can be maintained while adding new functionality.

9 References

[CP88] Cai, J., and Paige, R., "Program derivation by Fixed-Point Computation," *Science of Computer Programming Vol. 11, 1988/89, pp. 197-261.*

[D97] Dean, D. "The Security of Static Typing with Dynamic Linking," Proceedings of the Fourth ACM Conference on Computer and Communications Security, April, 1997. http://www.cs.princeton.edu/sip/

[C97] Cohen, R. "The Defensive Virtual Machine Specification 0.5," http://www.cli.com/software/djvm/index.html

[K73] Kildall, G. "A unified Approach to Global Program Optimization," POPL, 1973.

[K97] The Kimera project, http://kimera.cs.washington.edu/

[LY97] Lindholm, T. and Yellin, F. *The Java™ Virtual Machine Specification, Addison Wesley,* 1996.

[Mu97] Muchnick, S., *Advanced Compiler Design & Implementation,* Margan-Kaufmann, 1997.

[N97] Nipkow, T. and von Oheimb, D. "Java-light is Type-Safe – Definitely" To appear POPL98, http://www4.informatik.tu-muenchen.de/~nipkow/pubs/popl98.html

[Q97] Qian, Zhenyu "A Formal Specification of Java™ Virtual Machine Instructions," (Draft), http://www.informatik.uni-bremen.de/~qian/abs-fsjvm.html

[S97] Saraswat, V. "Java is not type safe," http://www.research.att.com/~vj/bug.html

[SJ95] Srinivas, Y. V. and Jüllig R., "Specware™: Formal Support for Composing Software," *Proceedings of the Conference on Mathematics of Program Construction,* Kloster Irsee, Germany, July 1995. Kestrel Institute Technical Report KES.U.94.5, http://www.kestrel.edu/HTML/publications.html

[Te74] Tennenbaum, "Automatic Type Analysis in a Very High Level Language," Thesis, New York University 1974.

[V97] Volpano, D., "A Type-Based Approach to Program Security," *Int'l Joint Conference on the Theory and Practice of Software Development, LNCS 1214,* Lille France, April 1997, pp. 607-621.