Toward a Provably-Correct Implementation of the JVM Bytecode Verifier

Alessandro Coglio Allen Goldberg Zhenyu Qian
Kestrel Institute
coglio, goldberg, qian@kestrel.edu

Abstract

This paper reports on our ongoing efforts to realize a provably-correct implementation of the Java Virtual Machine bytecode verifier. We take the perspective that bytecode verification is a data flow analysis problem, or more generally, a constraint-solving problem on lattices. We employ SPECWARE, a system available from Kestrel Institute that supports the development of programs from specifications, to formalize the bytecode verifier, and to formally derive an executable program from our specification.

1. Introduction

DoD applications are increasingly being implemented in distributed computing environments. These environments exacerbate security concerns, especially when mobile code is employed. Java provides *language-based mechanisms* that help address many security concerns. In particular, buffer overflow attacks, which account for as much as 50% of today's system vulnerabilities, exploit the absence of type safety in many languages, notably C and C++. Java is a type safe language and so eliminates this mode of attack. Furthermore, Java uses language-based mechanism for insuring correct program linking, and the enforcement of security policies.

In the Java language framework, Java source code is compiled to Java Virtual Machine *(JVM)* code, usually referred to as *bytecode*. It is bytecode rather than Java source that is transmitted as mobile code. The JVM cannot trust that this code is the unmodified output of a correct Java compiler. Thus, as part of the loading process the *JVM* verifies that the purported bytecode is valid *JVM* code. This verification procedure, performed by the *bytecode verifier*, is non-trivial. A major obejective is to establish the type safety of the code using data flow methods.

This paper reports on our ongoing efforts to realize a provably-correct implementation of the Java Virtual Machine bytecode verifier (or simply the *verifier*) from a formal specification using the SPECWARE™ System. SPECWARE [1], a system available from Kestrel Institute [2], supports the formal and provably-correct development of programs from specifications written in a specification notation based on high-order logic.

In previous papers [3, 4] we have specified the semantics of the *JVM* verifier. Collectively these papers deal with most aspects of the *JVM* including *JVM* subroutines, dynamic class loading, object initialization, interface types, arrays, and all primitive types. These papers take the perspective that bytecode verification is a *data flow problem*, or more generally, a *constraint-solving problem on lattices*. One advantage of this approach is that implementation of a bytecode verifier from such a specification can be derived as an instantiation of a generic algorithm for constraint solving.

In this paper, we describe our progress in formalizing the specifications in those papers using SPECWARE, and we describe the refinement methodology used to obtain an implementation.

This paper is organized as follows. The next section gives a detailed overview of our approach. In Section 3 we describe how our specification of the verifier is formalized in SPECWARE. In Section 4 we describe its refinement to a program using SPECWARE. In Section 5 we give a small example. This is followed by a description of related work and our conclusions.

2. Approach

2.1. Bytecode Verification, Data Flow Analysis, and Constraint Problems

Data flow analysis is a methodology used to establish assertions at program points that are invariant over all program executions. Because the types of local variables and stack elements vary during *JVM* execution, it is natural to view the bytecode verifier as a data flow problem. To specify a particular data flow problem, a control flow graph, a semilattice, an initial state, and transfer functions are specified. The semilattice captures the abstract program properties of interest, and transfer functions capture the behavior of *JVM* instructions with respect to the semilattice. The data flow framework includes algorithms that solve general data flow problems by fixed-point iteration. Theorems that assert algorithm termination, soundness and give a characterization of the accuracy of the solution have been proved [5] In particular, soundness and termination are assured if the semilattice has finite height and the transfer functions are monotone. In addition, if the

transfer functions are distributive, the algorithm yields the meet-over-all-paths solution, i.e. the sharpest or most accurate result possible. In our specification of the verifier, construction of a flow graph is trivial. The main challenge is to specify the semilattice and transfer functions.

In formalizing our specification, we chose a more general constraint framework [6] instead of data flow analysis. Let $L = \langle L, \sqsubseteq, \sqcap \rangle$ be a semilattice and F a collection of monotone functions of various arities over L. Let V be a collection of variable names. Let t denote a term formed from constants, $c \in L$, variables, $v \in V$, and function symbols from F. A *constraint solving problem* is a collection of *definite inequalities*, i.e. inequalities of the form $v \sqsubseteq t$ or $c \sqsubseteq t$. A solution is an assignment $I : V \rightarrow L$ satisfying each inequality. A solution M is *maximal* if for any solution I and any variable v, $I(v) \sqsubseteq M(v)$. In this paper, a reference to a *constraint solving problem* or, simply, a *CSP* refers to a problem of the described form.

It is not difficult to see that a data flow problem may be mapped to a *CSP* problem. For simplicity, assume each node of the control flow graph consists of a single *JVM* instruction. Let tf_i denote the transfer function formalizing the behavior of the instruction at node i.[1] Introduce a constraint variable, u_i for each node i of the control flow graph. For each edge (i, j) introduce the inequality $u_j \sqsubseteq tf_i(u_i)$.Our specification of the verifier generates *CSP*s of this form. Many of the properties enjoyed by the data flow architecture are also true of these *CSP* 's. A chaotic fixed-point iteration algorithm will converge to the maximal solution. The complexity of the algorithm is polynomial.

We chose to express the bytecode verifier as a *CSP* problem for the following reasons:

- We wish to explore the applicability of Kestrel-developed synthesis technology [7] that has been used to optimize a related class of constraint problems.
- This results in a robust specification that be can be modularly enhanced to perform other security-related static checks on bytecode, for example information flow analysis. More generally the bytecode verifier may be viewed as the verification condition generator of a proof carrying code implementation.

2.2. Some Salient Aspects of Our Bytecode Specification

The bytecode verifier determines if a *JVM* program is well typed. Because the methods in a class reference instance variables and methods defined in other classes, type consistency requires checking the *internal consistency* of a class, as well as its *external consistency* with referenced classes. Because class files are loaded dynamically, and because it is desirable to minimize constraints on when classes gets loaded, the verifier cannot assume that a referenced class has been loaded prior to verification of a referencing class. Thus, our specification maintains a *global typing context* consisting of *type assertions* derived from the declarations in a class, and *type assumptions* derived from references to external classes. The global typing context is one component of the semilattice.

Because we make no assumptions about the order that classes are loaded (so the least general common super-type of two object classes is generally not known when the class is verified), and because there is no greatest common super-type of two interface types, there is no meaningful *meet* operation definable for *JVM* reference types. Instead, we use a *set* to represent reference types. The intended meaning is that the static type of a reference is one of the (reference) types in the set. The set is a semilattice with union as the meet operation. Verification of the `invokevirtual` and other instructions add subtype assumptions to the global typing context.

2.3. Formalization of the Bytecode Verifier

2.3.1. Architecture of the verifier as a constraint problem.

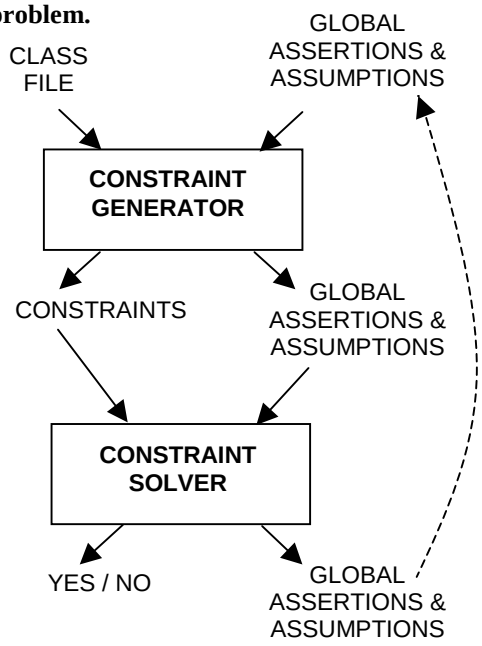

Figure 1: Verifier Architecture

Using the constraint approach, verification of a class file is performed in two steps, as illustrated in Figure 1. First, the global typing context is updated with assertions and assumptions derived from declarations in the constant

[1] If a *JVM* instruction raises an exception, its behavior differs from normal execution. Therefore, our actual specification associates transfer functions with edges, not nodes.

pool. Furthermore, a constraint problem is generated for each method defined in the class. In generating the constraint problem, it is assumed that the class file meets the static verification checks described in Section 4.9 of [8]. In the second phase, the constraint problem for each method is solved and the global typing context is updated with typing assumptions derived from the method code.

As defined above, a semilattice, a collection of monotone functions on the semilattice, and a set of inequalities parameterizes a *CSP*. The semilattice and monotone functions are defined once for the *JVM* — only the generated constraint inequalities depend on the method being verified.

2.3.2. Semilattice construction. We define a semilattice, L_{JVM}, that characterizes the information that the verifier maintains at each program point. This information includes the type of local variables and elements of the stack, as well as the global typing context, which includes assertions and assumptions about class declarations and subtype relationships, and the signature of referenced methods and instance variables. The type information regarding local variables and stack elements is not simply the static type of the entity, but holds information about the initialization status of objects, and other information needed to verify the proper use of the jsr and ret instructions.

We define L_{JVM} from some simple point and set semilattices using *semilattice-building operations*:

× takes two semilattices and forms their product;

seq takes a semilattice L and forms a semilattice of products (sequences) of elements from L;

⊕ takes two semilattices and forms their disjoint sum;

/ takes a semilattice and a suitable congruence relation and forms a semilattice whose elements are the equivalence classes induced by the relation. One use of this operation is to identify the bottom element of a binary product with the bottom elements of the component semilattices;

stk takes a semilattice L and forms a semilattice of bounded stacks whose elements are taken from L.

Note that these operations are generic semilattice constructions of utility beyond the *JVM*.

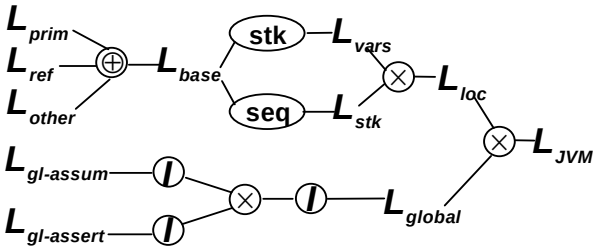

Figure 2: JVM Semilattice Construction

Figure 2 is a simplified view of the construction of L_{JVM} using the semilattice-building operations. In the figure, the ovals represent operations and unboxed text the names of the resulting semilattices. Thus, L_{base}, used to represent a stack element or local variable, is the (cascaded) disjoint sum of the three semilattices shown. As described above, the semilattice that represents a reference is a set semilattice of reference types. L_{base} is then used to form semilattices representing the stack and local variables. $L_{gl\text{-}assert}$ represents the set of global assertions. The quotient operation is applied to $L_{gl\text{-}assert}$ to construct a semilattice that identifies in a single equivalence class all inconsistent assertion sets and the bottom element of the semilattice.

2.4. Monotone Functions and Constraint Inequalities

Roughly speaking, our specification defines a monotone function for each transfer function corresponding to a *JVM* instruction. The transfer functions are constructed from monotone functions defined on component semilattices of L_{JVM}. For example, *push* and *pop* are defined on stack semilattices, proved to be distributive (hence monotone) functions, and used in the definition of transfer functions that manipulate the *JVM* stack. Most transfer functions are composed from constructor or destructor operations (like *push* and *pop*) of the semilattice-building operations (like *stk*(*L*)).

More precisely, because some transfer functions depend on the operand of the instruction, we actually defined families of parameterized transfer functions. For example, the transfer function for the `putfield` instruction is parameterized by the name of the object class containing the field and type of the referenced field.

Analysis of a method generates a constraint inequality of the form $u_j \sqsubseteq tf_i\ (u_i)$ for each edge *(i, j)* of the control flow graph. Constraints are represented as pairs of terms, using an abstract data type for terms. The constraint solving algorithm invokes a function *eval (t, e)* that evaluates a term *t* given an environment *e* that maps variables to semilattice values.

3. Formalization in SPECWARE

As mentioned in Section 1, SPECWARE is a system supporting the formal development of programs from specifications. Its core functionality is based on clear mathematical concepts from logic and category theory, and made accessible to the developer through a graphical user interface. A specification (*spec*) in SPECWARE is a theory in high-order logic. The system provides convenient mechanisms to build more complex specs out of simpler ones. One such mechanism is *instantiating* a *parameterized spec* (*pspec*): roughly, a pspec is a spec with an explicit "formal parameter" part, which upon instantiation gets "replaced" with an "actual parameter" spec.

Formalizing bytecode verification in SPECWARE along the lines described in Section 0, amounts to formalizing: the *JVM* semilattices; the transfer functions for the *JVM* instructions; the format of class files; the form of constraints; which constraints are derived from a given class file; and what is a (maximal) solution to a set of constraints. In this section, we provide an overview of the specs we developed for some of these concepts. To avoid cluttering this overview with non-substantial details, the examples we present are slight simplifications of the specs we actually wrote.

SPECWARE provides a library of specs for some popular concepts (e.g., sets, ordering relations, and arrays). Starting from some of them, we incrementally built our specs in a structured way, making extensive use of pspecs and instantiation, as well as of other composition mechanisms. We followed the rationale of "factorizing" common sub-concepts as much as possible, in order to produce more re-usable, readable, and elegant specs. In fact, many of the specs we wrote are completely independent of bytecode verification.

First of all, we wrote specs for (generic) semilattices, such as

```
spec SEMILATTICE is
 sort P
 op lq : P, P -> Boolean
 op meet : P, P -> P
 axiom reflexivity is
       lq(x,x)
 axiom anti-symmetry is
       lq(x,y) & lq(y,x) => x=y
 axiom transitivity is
       lq(x,y) & lq(y,z) => lq(x,z)
 axiom greates-lower-bound is
       lq(meet(x,y),x) &
       lq(meet(x,y),y) &
       (lq(z,x) & lq(z,y)=>
        lq(z,meet(x,y)))
end-spec
```

We wrote pspecs formalizing the construction of sequence semilattices, stack semilattices, and so on. For instance, we wrote a pspec `SEQUENCE-of-SEMILATTICE` having `SEMILATTICE` as formal parameter, and defining a new sort of tuples of semilattice points, and how the partial ordering and binary operation can be lifted to such tuples:

```
pspec SEQ-of-SEMILATTICE is
 parameter SEMILATTICE
 ...
 definition of meet :
              Seq, Seq -> Seq is
 axiom  meet (x,y) = z  <=>
        fa(i) comp(z,i) =
              meet (comp(x,i),
                    comp(y,i))
 ...
```

Analogously, we wrote pspecs `STACK-of-SEMILATTICE` (with *push* and *pop* operations), `QUOTIENT-of-SEMILATTICE`, etc. Next, we suitably instantiated them, starting from *JVM*-specific semilattices such as:

```
spec JVM-PRIMITIVE-SEMILATTICE is
 sort PrimSL
 const int : PrimSL
 const float : PrimSL
 const unusable : PrimSL
 ...
 definition of meet is
  axiom meet(int,float)= unusable
  axiom meet(int,int) = int
 ...
```

In order to formalize definite inequalities, terms over a semilattice with monotone functions must be formalized. Abstracting a little bit from that, we first wrote a spec `ALGEBRA` and a pspec `TERMS –over-ALGEBRA` having `ALGEBRA` as formal parameter:

```
spec ALGEBRA is
 sort Dom
 sort Fun
 op arity : Fun -> Nat
 op apply : Fun, DomList -> Dom
 ...
pspec TERMS-over-ALGEBRA is
 parameter ALGEBRA
 sort Term
 sort Var
 op const-term : Dom -> Term
 op var-term : Var -> Term
 op funapp-term :
    Fun, TermList -> Term
 sort Asg
 op asg-val : Asg, Var -> Dom
 op eval : Term, Asg -> Dom
 ...
```

Next, instantiating the carrier `Dom` to be a semilattice, adding axioms stating monotonicity for the elements in `Fun`, and pairing generic terms with constant terms or variable terms, we formalized definite inequalities, as well as *CSP*'s as sets of definite inequalities, and what is a (maximal) solution.

Our specs for transfer functions define a sort for them, and an `apply` operation to apply them to the *JVM* semilattice points. To avoid lengthy and repetitive definitions, we defined them as suitable compositions of some auxiliary functions. For instance, we defined a function which pops the top elements of a *JVM* stack semilattice point if they satisfy a "pattern" (e.g., the top two elements being both integers), and returns $\perp$ otherwise. Here is an excerpt:

```
spec TRANSFER-FUNCTIONS is
 sort TransFun
 op apply :
    TransFun, JvmSL -> JvmSL
 const iadd : TransFun
 ...
 axiom
  fa (...stk:StkSL...)
     apply(iadd,...stk...) =
     (...push (int,
               pop(pattern(int),
                   stk))...)
 ...
```

Clearly, by instantiating the definite inequality pspecs with the *JVM* semilattice and the transfer functions, we exactly obtained the spec for *JVM* constraint problems.

SPECWARE provides facilities to validate specs, by allowing the developer to enrich them with conjectures stating putative properties of the specs. The developer can then ask the system to *verify* a spec, which amounts to invoking a theorem prover (currently, Specware is connected to various theorem provers) to prove all the conjectures of the spec. In all our specs we included conjectures, stating for instance that the *JVM* primitive semilattice is really a semilattice, that a (generic) sequence semilattice is really a semilattice, and that our transfer functions are monotone:

```
...
theorem prim-reflexivity is
   fa(x:PrimSL) lq(x,x)
...
theorem seq-transitivity is
   fa(x,y,z:Seq)
     lq(x,y) & lq(y,z) => lq(x,z)
...
theorem transf-fun-monotonicity is
   fa(tf:TransFun, x,y:JvmSL)
     lq(x,y) =>
     lq (apply(tf,x),
         apply(tf,y))
...
```

4. Refinement in SPECWARE

In SPECWARE, programs are formally derived from specs by *refining* specs. Roughly, refining a spec amounts to "mapping" it into a new spec, which interprets the concepts of the initial one in terms of other concepts. These other concepts should be closer to those of some target executable language, and if they are sufficiently close, executable code can be generated by SPECWARE. Refinements can be sequentially composed, thus allowing code to be derived from specs through a series of successive steps. Furthermore, a refinement for a compound spec (e.g., an instantiated pspec) can be obtained from refinements for the individual components (e.g., for the pspec and for the actual parameter). Currently, SPECWARE can generate code for (functional subsets of) LISP and C++.

SPECWARE provides built-in mechanisms to represent constructed sorts (e.g., products, sums, and quotients) in target languages in terms of the representations of the component sorts. It also provides a library of refinements of common abstract structures (such as sets and bags) to more concrete structures (such as lists and arrays) which are "directly" representable in target languages. Starting from these mechanisms and refinements, we have refined our specs to LISP code. For instance, we refined the *JVM* primitive semilattice points to an enumeration of integers, with semilattice operations defined by cases:

```
...
definition of unusable : PrimSL is
 axiom  unusable = 1
definition of int : PrimSL is
 axiom  int = 2
definition of float : PrimSL is
 axiom  float = 3
...
definition of meet :
        PrimSL, PrimSL -> PrimSL is
 axiom ~(x=y) =>
       meet(x,y) = unusable
...
```

Sequence and stack semilattices have been refined to array and lists. Operations have been re-phrased to be constructive, as in:

```
definition of meet :
              Seq, Seq -> Seq is
  axiom  meet (x,y) =
         meet-aux (x, y, x, 1)
definition of meet-aux :
       Seq, Seq, Seq, Nat -> Seq is
  axiom geq(i,size(z)) =>
       meet-aux(x,y,z,i) = z
  axiom lt(i,size(z)) =>
       meet-aux(x,y,z,i) =
       meet-aux(x,y,
                change(z,i,
                  meet(comp(x,i),
                       comp(y,i))),
                succ(i))
```

An important refinement is to provide an actual algorithm to compute the maximal solution of a set of definite inequalities. We have in fact built and refined specs for the algorithm proposed in [6]. For example, a constraint of the form $u_3 \sqsubseteq tf_{\text{iadd}}(u_4)$ is represented in our generated LISP code as (roughly):

```
((VAR 3) (FUN-APP (TF 16) (VAR 4)))
```

And here is how the *meet* function over the *JVM* primitive semilattice is refined to LISP:

```
(DEFUN MEET-PRIM (X Y)
       (COND((NOT (= X Y)) 1)...))
```

We are going to further refine our specs for optimization, in order to generate more efficient code.

5. Example

Figure 3 below gives a method together with an explanation of each instruction. We assume that the method is contained in the class `C`. Note that in the instruction `putfield (Fld,D,C)`, `Fld` is the name of the field, `D` the type of the field and `C` the name of the class containing the field. Since program point `7` has two predecessors `5` and `6`, and the top stack entry may hold either the first or second actual parameter.

For the instructions in the example in Figure 3 we define the following transfer functions of type JvmSL $\rightarrow$ JvmSL:

$tf_{\textbf{aload}\ \text{ind}}$ (*asr*,*asm*,*var*,*stk*) :=
  if isRef (var_{ind}) *then* (*asr*,*asm*,*var*,*push*(*stk*, var_{ind}))
else $\perp$

$tf_{\textbf{if\_acmpeq}\ pp}$ (*asr*,*asm*,*var*,*stk*) :=
 if isRef (*top*(*stk*)) *and isRef* (*top*(*pop*(*stk*)))
 then (*asr*,*asm*,*var*,*pop*(*pop*((*stk*))) *else* $\perp$

$tf_{\textbf{goto}\ pp}(u)$:= *u*

$tf_{\text{putfield (Fld,D,C)}}$ (*asr*,*asm*,*vars*,*stk*) :=
 (*asr*,
 asm $\cup$ {*subtyping*(*top*(*stk*),*D*),
        subtyping(*top*(*pop*(*stk*)),*C*), `Fld`$\in$ *fields*(*C*)},
 vars, *pop*(*pop*(*stk*)))

$tf_{\textbf{return}}$ (*u*) := T

where T denotes an artificial top element in the semilattice *JvmSL*, and the function *subtyping* ({ref_1,…,ref_n}, *ref*') yields *true* if and only if each ref_i is a subtype of *ref*' in the *JVM*.

We view the instruction `return` as having a special final node as its successor program point. The head of each method has a special transfer function

head_tf (*asr*,*asm*) := (*asr*,*asm*,[*cnam*, ty_1, …, ty_m,
                $unus_{m+1}$,…, $unus_n$],[])

where *asr* and *asm* are given by an invoking site of the method, *cnam* is the class containing the declaration of the method, and ty_1, ..., ty_m are types of the parameters.

A constraint is created for each instruction. Let the instruction be at the program point *pp* and have a successor program point *pp'*. Then the constraint is of the form

$u_{pp'} \sqsubseteq tf\,(u_{pp})$

For the method head, a constraint

$u_0 \sqsubseteq$ *head_tf* (*cnam*, ty_1, …, ty_m, *asr*, *asm*)

is created at program point `0`, where *asr* and *asm* are given by an invoking site of the method.

```
Void m (J1, J2)          // The method has two arguments of interfaces J1 and J2
.limit local 3           // The method has 3 variables
                         // Set this-object and the actual parameters in the variables;
                         // set the empty stack
0 aload 0                // Load the object reference in variable 0 onto the stack.
1 aload 1                // Load the object reference in variable 1 onto the stack.
2 aload 2                // Load the object reference in variable 2 onto the stack.
3 if_acmpeq 6            // If the top entries in the stack are equal, then go to 6;
                         // else go to 4.
4 aload 1                // Load the object reference in variable 1 onto the stack.
                         // Go to 7.
5 goto 7                 // Load the object reference in variable 2 onto the stack.
7 putfield (Fld,D,c)     // Put the top stack entry into the field Fld of the object
                         // referenced by the second top stack entry
8 return                 // Terminates and returns.
```

Figure 3: A Simple Method

Program	*vars*	*stk*	*asm*
`void m(J1,J2)`			*asm* as input
`0 aload 0`	[C, *J*1, *J*2]	[]	*asm*
`1 aload 1`	[C, *J*1, *J*2]	[*C*]	*asm*
`2 aload 2`	[C, *J*1, *J*2]	[*C*, *J*1]	*asm*
`3 if_acmpeq 6`	[C, *J*1, *J*2]	[*C*, *J*1, *J*2]	*asm*
`4 aload 1`	[C, *J*1, *J*2]	[*C*]	*asm*
`5 goto 7`	[C, *J*1, *J*2]	[*C*, *J*1]	*asm*
`6 aload 2`	[C, *J*1, *J*2]	[*C*]	*asm*
`7 putfield (Fld,D,C)`	[C, *J*1, *J*2]	[*C*, {*J*1, *J*2}]	*asm*
`8 return`	[C, *J*1, *J*2]	[]	*asm* $\cup$ {*subtyping* ({*J*1, *J*2}, *D*), *Fld* $\in$ *fields*(*C*)}}

Figure 4: Legal Location Types for the Method in Figure 3

Figure 4 shows the maximal solution to the constraint inequalities generated for the method code. Note that at program point `7`, the semilattice value of the top entry of the stack is a set with two elements, since its static type is either the static type of the first or second actual parameter of the method. In all other cases where a stack or local variable holds a reference type, the set of possible types is a singleton. To simplify, we have suppressed braces around singleton sets. The constraint *subtyping*({*J*1, *J*2}, *D*) in the *asm* component at program point `8` ensures that *D* is a superinterface of `J1` and `J2`.

6. Related Work

Bertelsen formalized *JVM* instructions using state transitions [9]. Cohen described a formal semantics of a subset of the *JVM*, but runtime checks are used to assure type-safe execution [10]. Both approaches did not consider static type checking, thus are not directly relevant to bytecode verification.

Stata and Abadi [11] proposed a type system for subroutines, provided lengthy proofs for the soundness of the system and clarified several key semantic issues about subroutines.

Qian [3] presented a constraint-based typing system for objects, primitive values, methods and subroutines and proved the soundness.

Freund and Mitchell [16] made a significant extension of Stata and Abadi's type system [11] by considering object initialization.

Hagiya and Tozawa [12] presented another type system for subroutines, where the soundness proof is extremely simple. Hagiya and Tozawa discussed issues relating to implementation of their type system, but they did not formally describe their implementation. In fact, since they did not consider objects, their implementation did not address many of the issues that we have.

Pusch [13] formalized a subset of the *JVM* in the theorem prover Isabelle/HOL based on the work by Qian [3], thus achieving a high level of assurance. All of this work is basically aimed at achieving a sound specification, but did not consider how to develop a provably correct implementation.

Goldberg [4] directly used data flow analysis to formally specify bytecode verification focusing on type-correctness and global type consistency for dynamic class loading. He successfully formalized a way to relate bytecode verification and class loading.

Saraswat [14] studied static type-(un)safety of JAVA in the presence of more than one class loader. We do not consider class loaders in this paper.

The Kimera project [15] was quite effective in detecting flaws in commercial bytecode verifiers. Using a comparative testing approach, they wrote a reference bytecode verifier and tested commercial bytecode verifiers against it. A particularly interesting point was that their code is well structured and organized, and derived from the English *JVM* specification. It achieves a higher level of assurance than commercial implementations. However, since there is no formal specification, it is not possible to reason about it, or establish its formal correctness.

7. Conclusions and Future Work

We have specified and implemented via refinement nearly all aspects of the bytecode verifier. Our specification omits treatment of exceptions and the use of two instructions: `jsr` and `ret`. While these instructions add significant complexity to the bytecode verifier, we

have previously formalized their semantics and expect their specification and refinement to be a straightforward extension of our current verifier. Performance of our generated code is sufficient to serve as a reference implementation the verifier.

We plan to compare the results from our derived verifier with other bytecode verifiers. This will contribute to increasing our assurance that the specification captures the intended semantics of the informal specification and may expose errors in these other verifiers.

We are currently using a similar approach to specify another security-critical component of the JVM, the class loader.

8. References

[1] Srinivas, Y.V. and R. Jüllig. Specware™: Formal Support for Composing Software. In, *Proceedings of the Conference on Mathematics of Program Construction*, B. Moeller, Ed. Berlin: Springer-Verlag, 1995, pp. 399--422. Lecture Notes in Computer Science, Vol. 947.

[2] *Kestrel Institute KEEP Program. http://www.kestrel.edu/HTML/keep.html/.*

[3] Qian, Z. A Formal Specification of Java™ Virtual Machine Instructions for Objects, Methods, and Subroutines. In, *Formal Syntax and Semantics of Java™*, J. Alves-Foss, Ed. Berlin: Springer-Verlag, LNCS #1523, 1998, pp. 271-312..

[4] Goldberg, A. A Specification of Java Loading and Bytecode Verification. In, *Proceedings, 5th ACM Conference on Computer and Communications Security*. San Francisco, CA., November 1998. ACM Press.

[5] Muchnick, S. *Advanced Compiler Design and Implementation*. San Francisco, CA: Morgan Kaufmann, 1997.

[6] Rehof, J. and T. ÆMogensen. Tractable Constraints in Finite Semi-lattices. In, *Third International Static Analysis Symposium (SAS)*, 1996, pp. 285-330. Springer-Verlag.

[7] Westfold, S.J. and D.R. Smith. *Synthesis of Efficient Constraint Satisfaction Programs.* Kestrel Institute Tech. Rep., 1998. .

[8] Lindholm, T. and F. Yellin. *The Java™ Virtual Machine Specification.* Reading, MA: Addison-Wesley, 1996.

[9] Bertelsen, P. *Semantics of java byte code*. Copenhagen: Royal Veterinary and Agricultural University Tech. Rep., 1997. *http://www.dina.kvl.dk/~pmb/.*

[10] Cohen, R.M. *The Defensive Java Virtual Machine Specification.* Computational Logic, Inc. Tech. Rep., 1997. .

[11] Stata, R. and M. Abadi. A type system for Java bytecode subroutines. In, *Proceedings, 25th ACM Symposium on the Principles of Programming Languages*. San Diego, CA, January 1998. ACM Press.

[12] Hagiya, M. and A. Tozawa. On a New Method fot Dataflow Analysis of Java Virtual Machine. In, *Proceedings of the 1998 Static Analysis Symposium*, 1998. (To appear).

[13] Pusch, C. *Formalizing the Java Virtual Machine in Isabelle/HOL*. Technische Unversität München Tech. Rep. TUM I9816, October 18, 1998. *http://www4.informatik.tu-muenchen.de/~isabelle/bali/.*

[14] Saraswat, V. *Java is not Type-safe*. AT&T Research Tech. Rep., 1997. .

[15] Sirer, E.G., S. McDirmid, and B. Bershad. *A Java System Security Architecture*. University of Washington Tech. Rep., 1997.

[16] Freund, S. and J. Mitchell. *A Type System for Object Initialization in the Java Bytecode Language*. in *Proceedings of OOPSLA'98*. October 1998. Vancouver, B.C., Canada. ACM Press.

.