Toward a Provably-Correct Implementation of the JVM Bytecode Verifier

Alessandro Coglio Allen Goldberg Zhenyu Qian

Kestrel Institute,
3260 Hillview Avenue
Palo Alto, CA 94304, USA
{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 dataflow 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

This paper, reports on our ongoing efforts to realize a provably-correct implementation of the Java Virtual Machine (*JVM*) bytecode verifier (or simply the *verifier*) from a formal specification using the SPECWARE System. SPECWARE [SJ95], a system available from Kestrel Institute [KES], supports the formal and provably-correct development of programs from specifications written in a specification notation based on high-order logic.

In previous papers [Qia98, Gol98] 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 *dataflow 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 §3 we describe how our specification of the verifier is formalized in SPECWARE. In §4 we describe its refinement to a program using SPECWARE. In §5 we give a small example. This is followed by a description of related work and our conclusions.

2 Approach

2.1 Bytecode Verification, Dataflow Analysis, and Constraint Problems

Dataflow 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 dataflow problem. To specify a particular dataflow 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 dataflow framework includes algorithms that solve general dataflow problems by fixed-point iteration. Theorems that assert algorithm termination, soundness and give a characterization of the accuracy of the solution have been proved [Muc97]. 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 [RM96] instead of dataflow 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. inequalties 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 dataflow 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 dataflow 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 [JS98] that has been used to optimize a related class of constraint problems.
- We may implement the verifier using Bane [FAFS98, FFA98, BAN], an an off-the-shelf constraint solver designed specifically for program analysis.
- Our work formalizing *CSP* in Specware is more general and can be applied to problems other than just dataflow analysis.

A disadvantage of formulating the verifier as a *CSP* rather than a dataflow problem is a loss of efficiency due to the generality of *CSP*. In a dataflow problem the control flow graph explicates those constraints that are violated when a lattice value associated with a node is updated. However, the program refinement technology described in [JS98] can restore this efficiency.

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

[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.

minimize constraints on when classes gets loaded, the verifier cannot assume that a referenced class has been loaded prior to its verification. 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 unique least general common superclass of two interface types, there is no meaningful *meet* operation definable for *JVM* reference types. Instead, we use *sets* to represent reference types. The intended meaning is that the static type of a reference is one of the (reference) types in the set. These sets form a semilattice with union as the meet operation. Verification of the `invoke_virtual` 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

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 pool. Furthermore, a constraint problem is generated for each method defined in the class. In generating the constraint problem, it is checked that the class file meets the static verification checks described in Section 4.9 of [LY96]. 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 *CSP* is parameterized by a lattice, a collection of monotone functions on the lattice, and a set of inequalities. The lattice and monotone functions are defined once for the *JVM*— only the generated constraint inequalities depend on the method being verified.

2.3.2 Lattice 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

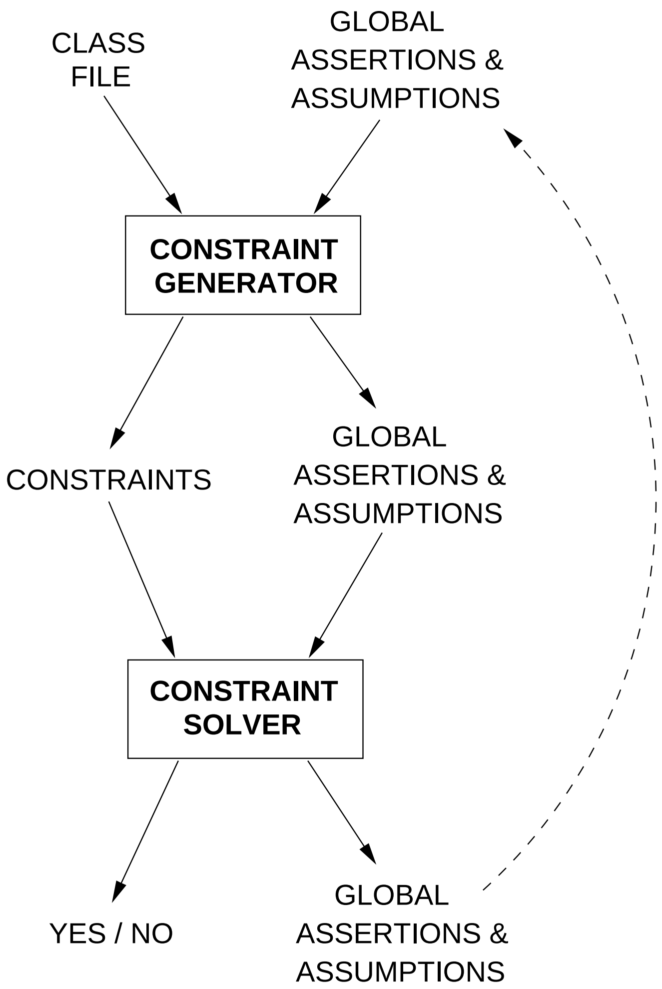

Figure 1: Verifier Architecture

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 lattices using *lattice building operations*:

$\times$ takes two semilattices and forms their product;

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

$\oplus$ 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 these operations are generic lattice constructions of utility beyond the *JVM*.

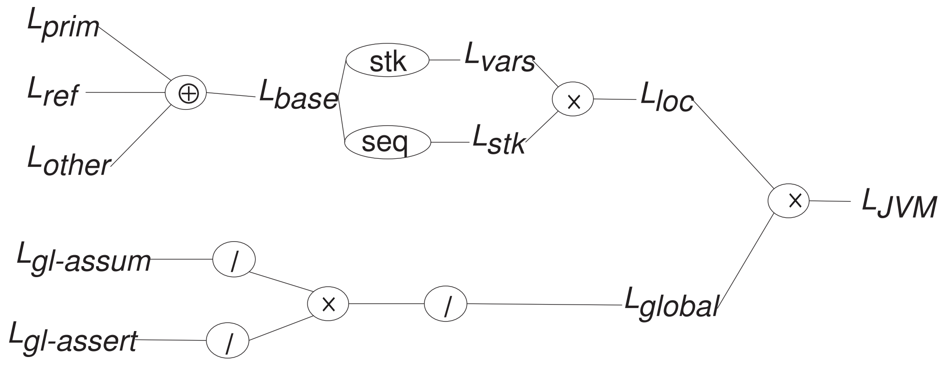

Figure 2: *JVM* Lattice Construction

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

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 lattices 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 lattice building operations (like *stk(L)*).

More precisely, because some transfer functions depend on the operand of the instruction, we actually defined parameterized families of 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 lattice values.

3 Formalization in Specware

As mentioned in §1, SPECWARE is a system supporting the formal development of programs from specifications. Its core functionalities are based on clear mathematical concepts from logic and category theory, and are 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 §2, 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, 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 r : P, P -> Boolean
 op f : P, P -> P
 axiom reflexivity is  r(x,x)
 axiom anti-symmetry is  r(x,y) & r(y,x) => x=y
 axiom transitivity is  r(x,y) & r(y,z) => r(x,z)
 axiom f-extr-bound-of-args is
         r(f(x,y),x) & r(f(x,y),y) &
         (r(z,x) & r(z,y) => r(z,f(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 SEQUENCE-of-SEMILATTICE is
 parameter SEMILATTICE
 ...
 definition of f : Seq, Seq -> Seq is
 axiom  f (x, y) = z  <=>
        forall(i) comp(z,i) = f (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[2]:

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

Next, instantiating the carrier `Crr` 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

[2] In the specs below, `CrrList` and `TermList` are sorts for finite lists of carrier elements (of sort `Crr`) and terms (of sort `Term`), respectively. `Asg` is a sort for assignments, i.e. finite maps from variables (of sort `Var`) to carrier elements.

as suitable compositions of some auxiliary functions. For instance, we defined a function *pop-match* which pops the top element of a *JVM* stack semilattice point if it "matches" a specified type (e.g., if it is an integer), and returns $\perp$ (*bottom*) otherwise. Here is an excerpt[3]:

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

Clearly, by instantiating the definite inequality pspecs with the *JVM* semilattice and the transfer functions, we exactly obtain 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, KITP [WG94]) 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  forall(x:PrimSL) leq(x,x)
...
theorem seq-transitivity is
 forall(x,y,z:Seq) r(x,y) & r(y,z) => r(x,z)
...
theorem transf-fun-monotonicity is
 forall(tf:TransFun, x,y:JvmSL)
   leq(x,y) => leq (apply(tf,x), apply(tf,y))
...
```

[3]In the spec below, `TransFun` is a sort for transfer function names, and `JvmSL` is a sort for the whole *JVM* semilattice points (i.e., including assertion, assumption, local variable, and stack semilattice points).

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, 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 are currently refining our specs to LISP code. For instance, we are refining the *JVM* primitive semilattice points to an enumeration of integers, with semilattice operations defined by cases:

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

Sequence and stack semilattices are being refined to arrays and lists. Operations are being re-phrased to be constructive, as in:

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

```
f-aux(x,y,change(z,i,f(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 are in fact building and refining specs for the algorithm proposed in [RM96].

For example, a constraint of the form $u_3 \sqsubseteq tf_{\mathtt{iadd}}(u_4)$ is represented in our generated LISP code as:

```
((VAR 3) ((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 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 declaring the field. Since the program point `7` has two predecessors `5` and `6`, 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

```
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.
5 goto 7                 // Go to 7.
6 aload 2                // 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

type $JvmSL \rightarrow JvmSL$:

$$
\begin{aligned}
&tf_{\texttt{aload}\ ind}(asr, asm, var, stk) := \\
&\quad \textit{if } isRef(var_{ind}) \textit{ then } (asr, asm, var, push(stk, var_{ind})) \textit{ else } \bot \\
&tf_{\texttt{if\_acmpeq}\ pp}(asr, asm, var, stk) := \\
&\quad \textit{if } isRef(top(stk)) \textit{ and } isRef(top(pop(stk))) \\
&\quad \textit{then } (asr, asm, var, pop(pop((stk))) \textit{ else } \bot \\
&tf_{\texttt{goto}\ pp}(u) := u \\
&tf_{\texttt{putfield(Fld,D,C)}}(asr, asm, var, stk) := \\
&\quad (asr, \\
&\quad asm \cup \{subtyping(top(stk), D), subtyping(top(pop(stk)), C), Fld \in fields(C)\}, \\
&\quad vars, pop(pop(stk))) \\
&tf_{\texttt{return}}(u) := \top
\end{aligned}
$$

where $\top$ denotes an (artificially added) greatest element in the *JVM* semi-lattice *JvmSL*, and $subtyping(\{\overline{ref_n}\}, ref')$ is an assumption stating that each ref_i with $1 \leq i \leq n$ is a subtype of ref' in the JVM.

We view the instruction `return` as having a special final node as its successor program point.

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

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

The head of each method has a special transfer function

$$head\_tf(cnam, \overline{ty_m}, asr, asm) := (asr, asm, [cnam, \overline{ty_m}, unus_{m+1}, \cdots, unus_n], [])$$

where asr and asm are the current global assertions and assumptions, $cnam$ is the class containing the declaration of the method, and $\overline{ty_m}$ 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

$$u_0 \sqsubseteq head\_tf(cnam, \overline{ty_m}, asr, asm)$$

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

Figure 4 shows the maximal solution to the constraint inequalities generated for the method code. Note that at the program point `7`, the lattice 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 Figure 4, we have supressed braces around singleton sets. The constraint $subtyping(\{J1, J2\}, D)$ in the asm component at `8` assures that D is a superinterface of `J1` and `J2`.

6 Related work

Bertelsen formalized JVM instructions using state transitions [Ber97]. Cohen described a formal semantics of a subset of the JVM, but runtime checks are used to assure type-safe execution [Coh97]. Both approaches did not consider static type checking, thus are not directly relevant to bytecode verification.

Stata and Abadi [SA98] proposed a type system for subroutines, provided lengthy proofs for the soundness of the system and clarified several key semantic issues about subroutines. Freund and Mitchell [FM98] made a significant extension of Stata and Abadi's type system by considering object initialization. Hagiya and Tozawa [HT98] presented another type system for subroutines, where the soundness proof is extremely simple. Qian [Qia98] presented a constraint-based typing system for objects, primitive values, methods and subroutines and proved the soundness. Pusch [Pus98] formalized a subset of the *JVM* in the theorem prover Isabelle/HOL 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. Note that 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 the issues that we have.

Goldberg [Gol98] directly used dataflow 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 [Sar97] 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 [SMB97] 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. 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

This work is ongoing. We expect to generate LISP code for a significant subset of the verifier over the next few months. We are concentrating first on generating code for the constraint solver. Initially, the constraint generator will be written by hand. We do not expect the derived verifier to be very efficient. Partial evaluation should be a very effective optimization method since much of the code will alternatively construct and destruct the multiple layers of component lattice structure. We also plan to extend the specification to cover all aspects of the verifier.

We are also studying implementing the verifier using the BANE constraint solver. In this scenario, the constraint generator will generate a constraint problem in the BANE constraint language. The constraint language used by BANE is closely related to the *CSP* scheme used here. The lattice used by BANE is a set lattice over regular trees. Most of the lattice constructions, including sums, binary and sequences, and stacks can be modeled using BANE. The BANE simplifier then replaces the derived constraint solver.

Recently, Qian [Qia97] presented a dataflow analysis algorithm (scheme), that non-deterministically uses formal typing rules to compute the smallest types for memory locations of *JVM* program. He rigorously proved the correctness, termination and completeness of the algorithm. He paid special attention to subroutines and objects. Additional work is needed to see if that scheme can be expressed in the *CSP* framework used here. Work on this is well under the way.

We also plan to compare the results from our derived verifier with other bytecode verifiers. All of these activities will contribute to increasing our assurance that the specification and implementation are correct.

References

[BAN] The Berkeley ANalysis Engine (BANE), `http://www.cs.berkeley.edu/Research/Aiken/bane.html`.

[Ber97] Peter Bertelsen. Semantics of java byte code. `http://www.dina.kvl.dk/~pmb/`, 1997.

[Coh97] R. M. Cohen. The Defensive Java Virtual Machine specification. Technical report, Computational Logic inc., 1997.

[FAFS98] Manuel Fähndrich, Alexander Aiken, Jeffrey S. Foster, and Zhendong Su. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the 1998 Conference on Programming Languages Design and Implementation*, Montreal, June 1998.

[FFA98] Manuel Fähndrich, Jeffrey S. Foster, and Alexander Aiken. Tracking down exceptions in standard ml programs. Technical report, UC Berekley, Feb 1998. UCB Computer Science Technical Report.

[FM98] Stephen Freund and John Mitchell. A type system for object initialization in the java bytecode language (summary). *Electronic Notes in Theoretical Computer Science*, 10, 1998. `http://www.elsevier.nl/locate/entcs/volume10.html`.

[Gol98] Allen Goldberg. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conference on Computer and Communications Security*, 1998. To appear.

[HT98] Masami Hagiya and Akihiko Tozawa. On a new method fot dataflow analysis of Java Virtual Machine subroutines. In *Proc. 1998 Static Analysis Symposium*. Springer-Verlag LNCS, 1998. To appear.

[JS98] Stephen J.Westfold and Douglas R. Smith. Synthesis of efficient constraint satisfaction programs. Technical report, Kestrel Institute, April 1998.

[KES] Kestrel Institute Keep Program, `http://www.kestrel.edu/HTML/keep.html/`.

[LY96] Tim Lindholm and Frank Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, 1996.

[Muc97] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, 1997.

[Pus98] C. Pusch. Formalizing the Java Virtual Machine in Isabelle/HOL. Technical report, TUM I9816, Technische Unversität München, 1998. `http://www4.informatik.tu-muenchen.de/~isabelle/bali/`.

[Qia97] Zhenyu Qian. Constraint-based specification and dataflow analysis for Java™ byte code verification. Technical report, Kestrel Institution, 1997. `http://www.kestrel.edu/~qian/abs-jvmdflow`, to appear.

[Qia98] Zhenyu Qian. A formal specification of Java™ virtual machine instructions for objects, methods and subroutines. In Jim Alves-Foss, editor, *Formal Syntax and Semantics of Java™*. Springer Verlag LNCS, 1998. To appear.

[RM96] Jakob Rehof and Torben Æ. Mogensen. Tractable constraints in finite semi-lattices. In R. Cousot and D. A. Schmidt, editors, *Third International Static Analysis Symposium (SAS)*, volume 1145 of *Lecture Notes in Computer Science*, pages 285–30. Springer, September 1996.

[SA98] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Proc. 25st ACM Symp. Principles of Programming Languages*, 1998.

[Sar97] V. Saraswat. Java is not type-safe. Technical report, AT&T Research, 1997. `http://www.research.att.com/~vj/bug.html`.

[SJ95] Y. V. Srinivas and Richard Jüllig. Specware: Formal support for composing software. In B. Moeller, editor, *Proceedings of the Conference on Mathematics of Program Construction*, pages 399–422. LNCS 947, Springer-Verlag, Berlin, 1995.

[SMB97] Emin Gün Sirer, Sean McDirmid, and Brian Bershad. A Java system security architecture. `http://kimera.cs.washington.edu/`, 1997.

[WG94] T. C. Wang and Allen Goldberg. KITP-93: An automated inference system for program analysis. In A. Bundy, editor, *Proceedings of 12th Conference on Automated Deduction*, pages 831–836. Springer-Verlag, Berlin, 1994. Lecture Notes in Artificial Intelligence, Vol. 814.