Interprocedural Analysis for JVML Verification *

William Retert and John Boyland

University of Wisconsin-Milwaukee, USA,
{williamr,boyland}@cs.uwm.edu

Abstract. Some of the problems encountered in the verification of subroutines in the Java Virtual Machine Language (JVML) are similar to problems already solved for interprocedural analysis in high level languages. This naturally leads to the question of whether techniques for the latter may be successfully applied to the former. To this end, we apply a general framework for interprocedural abstract interpretation to JVML0, a subset of JVML that isolates subroutines.

1 Introduction

Java bytecode verification is an important and oft-studied problem. Formalizations of verification tend to rely on some form of dataflow analysis or equivalent abstract interpretation to find satisfactory abstract values or types for local variables and the operand stack. These analyses are complicated by the presence of subroutines in the Java Virtual Machine Language (JVML).

JVML subroutines are a low-level mechanism for code reuse; their use is generally confined to the compilation of `finally` clauses, which must otherwise be included after each branch of a try ... catch. They provide several interesting challenges to analysis. First, as low-level mechanisms, they provide little in the way of inherent structure. Return addresses are pushed onto the operand stack by the `jsr` instruction, while the `ret` x instruction returns to any address stored in variable x. This problem is excaberated by other instructions potentially transferring control out of the current subroutine. Second, subroutines must be polymorphic over variables not affected by that subroutine, directly or indirectly.

The problem of verifying JVML subroutines has been essentially solved. The first difficulty is generally addressed either by treating subroutine calls

* Work supported in part by the National Science Foundation (CCR-9984681) and the Defense Advanced Research Projects Agency and Rome Laboratory, Air Force Materiel Command, USAF under contract F30602-99-2-0522. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation, Defense Advanced Research Projects Agency, Rome Laboratory, or the U.S. Government.

and returns as branches and analyzing the Java method "globally," or by analyzing each subroutine call separately, effectively inlining all subroutines. The former tends to use some form of polymorphic types for the unaffected variables, while the latter avoids the problem by duplicating all variables for each call, affected or not.

These two approaches have also been used in interprocedural analysis for high-level languages. Treating procedure calls as edges in the control-flow graph, which groups all calls into a single representation, is convenient, but loses information. Representing each call separately makes the analysis context sensitive and gives it a higher degree of precision; however, repeatly analyzing the same procedure is inefficient, and in the presence of recursion, impossible. Therefore, many analyses have sprung up which attempt to bridge the gap by extracting those portions of the context which are relevant to the called procedure. We have designed a framework in which such analyses can be easily formulated and proven correct.

Analyses extracting relevant context seem applicable to JVML verification, as they immediately allow the separation of unaffected variables as irrelevant. However, the lack of structured control flow poses interesting challenges for a framework designed with high-level languages in mind. In this paper, we describe both our framework and an initial application of our framework to the problem of analyzing JVML subroutines. In particular, we examine the simple language JVML0, altered to allow recursion.

This analysis provides an alternate approach to verifying subroutines. It also demonstrates the utility of our framework, which we would like to apply to problems in higher-level languages with structured control flow, but also more complex abstract values and less clear notions of relevance.

2 A Framework for Interprocedural Analysis

Following Mycroft [8], we say that a "program" consists of a sequence of mutually recursive procedures. The exact nature of the composition of procedures will be ignored for now. We define a single construct, an "interpretation" to cover both standard and abstract semantics [2], and thus a program has no semantics of itself. Two semantics may be related in the style of Nielson, Nielson and Hankin's abstraction framework [9].

Definition 1. *A* program *P is a sequence of n procedures $p_i, 0 < i \leq n$. An* interpretation *of a program $\mathscr{I} = (\mathscr{S}, \mathscr{F}_i \mid 0 < i \leq n)$ is a lattice representing program state and a sequence of functors on that state lattice to implement the procedures, where each functor maps n state transformers to a new state transformer: $\mathscr{F}_i : (\mathscr{S} \xrightarrow{\text{m}} \mathscr{S})^n \to (\mathscr{S} \xrightarrow{\text{m}} \mathscr{S})$ where $\mathscr{S} \xrightarrow{\text{m}} \mathscr{S}$ is the set of all state transformers, that is, monotonic functions on the state lattice. The* meaning *of a program in this interpretation (written $P_{\mathscr{I}}$) is the least fixed point solution to the equation:*

$$(f_1, \ldots, f_n) = (\mathscr{F}_1, \ldots, \mathscr{F}_n)(f_1, \ldots, f_n)$$

The meaning always exists if $\mathscr{S}$ is a complete lattice and the functors are themselves monotonic on the (complete) functional lattice $\mathscr{S} \xrightarrow{\text{m}} \mathscr{S}$.

Essentially, each $\mathscr{F}_i$ represents an (abstract) interpretation of the ith procedure, which uses $(f_1, \ldots, f_n)$ to approximate procedure calls internally. Any analysis closed over these functors will be a provably safe approximation of their least fixed point.

The approximations we substitute into functors consist of pairs of input and output states. The complete set of all such pairs will describe the behavior of the procedure perfectly. Unfortunately, that complete set is typically infinite. We introduce what we call parameterization to extend a finite set of pairs to approximate all possible input states.

Definition 2. *A lattice L is* parameterized using $(D, \phi, \otimes)$*, if D is an arbitrary set, and ϕ and $\otimes$ are functions such that ϕ maps a pair of values from the set and the lattice to the lattice ($\phi : L \times D \to L$), $\otimes$ maps a pair of values from the set back to the set ($\otimes : D \times D \to D$), and they satisfy the following properties for all $v, w \in L, S \subseteq L, d, d' \in D$:*

$$v \sqsubseteq w \implies \phi(v,d) \sqsubseteq \phi(w,d) \qquad \text{(monotonic)}$$

$$\phi(\bigsqcap_{v \in s} v, d) = \bigsqcap_{v \in s} \phi(v,d) \qquad \text{(completely distributive w.r.t. } \sqcap\text{)}$$

$$\phi(\phi(v,d), d') = \phi(v, d \otimes d') \qquad \text{(}\phi\text{-}\otimes \text{ connection)}$$

Monotonicity ensures that the parameter can be independently varied. Distributivity with regard to $\sqcap$ allows us to express the meaning of a function with a set of pairs (as in a conjunctive type system) and completeness makes it easier to define what those pairs mean. The identity connecting ϕ and $\otimes$ ensures that we can compose uses of ϕ. One may think of it as a closure property of D.

Example 1. Let L be the lifted lattice of sequences of lifted integers:

$$\begin{aligned} L &= (\mathbf{Z}_\bot^\top)^{*\,\top}_\bot \\ &= \{\bot, \top, \langle\rangle, \langle\bot\rangle, \langle\top\rangle, \langle 0\rangle, \langle 1\rangle, \langle -1\rangle, \ldots, \langle\bot,\bot\rangle, \ldots, \langle 2,3\rangle, \ldots, \langle\bot,-4,\top\rangle, \ldots\} \end{aligned}$$

Sequences of the same size may be ordered:

$$x_i \sqsubseteq y_i \forall_{1 \le i \le n} \implies \bot \sqsubseteq \langle x_1, \ldots, x_n\rangle \sqsubseteq \langle y_1, \ldots, y_n\rangle \sqsubseteq \top$$

Then L is parameterized using $(L, \cdot, \cdot)$ where $\cdot$ is sequence concatenation made strict in $\bot$ and $\top$:

$$\begin{aligned} \langle x_1, \ldots, x_n\rangle \cdot \langle y_1, \ldots, y_m\rangle &= \langle x_1, \ldots, x_n, y_1, \ldots, y_m\rangle \\ \langle \ldots\rangle \cdot \top = \top \cdot \langle \ldots\rangle &= \top \\ \langle \ldots\rangle \cdot \bot = \bot \cdot \langle \ldots\rangle &= \bot \\ \bot \cdot \top = \top \cdot \bot &= \bot \end{aligned}$$

The previous example represents the state of a program as a sequence. This makes sense for a pure stack machine, but often one has variables as well. The following example extends the state with variables. The parameterization function allows irrelevant variables to be parameterized away and then added back later.

Example 2. Let L be an arbitrary lattice of values and V be a finite set of variables. Let $V \mapsto L$ be the set of finite partial functions from V to L. We order two functions f and g if f is at least as precise as g for all elements in g's domain:

$$\forall_{x \in Dom\, g} f(x) \sqsubseteq g(x)$$

We define a (non-commutative) operation $+$ on functions, so that $f+g$ is f extended with any bindings that g has outside the domain of f:

$$f + g = f \cup (g - (Dom\, f \times L))$$

Then $S = (L^* \times (V \mapsto L))^{\top}_{\bot}$ is parameterized by (S, ϕ, ϕ) where

$$\begin{aligned} \phi((l,f),(l',f')) &= (l \cdot l', f + f') \\ \phi(\top, s) = \phi(s, \top) &= \top \\ \phi(\bot, s) = \phi(s, \bot) &= \bot \end{aligned}$$

Suppose we have a monotonic function f for which $f(v) = v'$, and thus $w \sqsubseteq v \implies f(w) \sqsubseteq v'$. Parameterization increases the usefulness of this input-output pair. We may have a w that is incomparable with v, or simply one that is much more precise than v. In the first case, we can't use the pair at all, and in the second case we lose "too much" precision. Thus instead we find a d such that $w = \phi(v,d)$. This d represents information about w that is not captured by v. We use the input-output pair by using $\phi(v',d)$ for $f(w)$. In essence, we abstract away part of the state, perform the call, and then re-attach this state. This operation is sound if the function does the same or better when given the complete lattice value versus when given only "part" of it. This intuition is captured in the following definition which gives the set of such functions:

Definition 3. *For a lattice L parameterized by $(D, \phi, \otimes)$, let H_ϕ be the set of functions* homomorphic with respect to ϕ*, that is the subset of functions $f \in (L \xrightarrow{\text{m}} L)$ where for all $v \in L, d \in D$:*

$$f\,\phi(v,d) \sqsubseteq \phi(f\,v, d) \qquad \text{(homomorphic w.r.t. } \phi\text{)}$$

The set H_ϕ is closed over four simple functors (composition, join, meet, and indefinite iteration).

Example 3. Using the parameterization from Example 1, the function dup is homomorphic with respect to sequence concatenation, where:

$$\begin{aligned} \mathsf{dup}\,\langle x, \ldots \rangle &= \langle x, x, \ldots \rangle\ , \\ \mathsf{dup}\,\langle\rangle = \mathsf{dup}\,\top = \top, \quad \mathsf{dup}\,\bot &= \bot\ . \end{aligned}$$

On the other hand, the function length which pushes on the stack the former length of the stack is not homomorphic w.r.t. $\cdot$, where:

$$\begin{aligned} \mathsf{length}\,\langle x_1,\ldots,x_n\rangle &= \langle n,x_1,\ldots,x_n\rangle\ , \\ \mathsf{length}\,\top = \top\ , \quad \mathsf{length}\,\bot &= \bot\ . \end{aligned}$$

The function fails to be homomorphic because the result depends incomparably (in terms of the lattice) on the sequence length.

Example 4. Given the parameterization with variables from Example 2, the following functions are homomorphic with respect to the ϕ function:

$$\begin{aligned} \mathsf{load}\,v\,(l,f) &= (\langle f(v)\rangle\cdot l, f) \quad (f(v)\text{ defined}) \\ \mathsf{load}\,v\,(l,f) &= (\langle\top\rangle\cdot l, f) \quad (\text{otherwise}) \\ \mathsf{store}\,v\,(\langle x\rangle\cdot l,f) &= (l,\{(v,x)\}+f) \quad (f(v)\text{ defined}) \\ \mathsf{store}\,v\,(l,f) &= \top \quad (\text{otherwise}) \end{aligned}$$

where load and store are strict in $\top$ and $\bot$ as usual.

The obvious "simplification" of completing all functions by giving a default value of $\top$ would make the use of $+$ in the definition of ϕ pointless. It would not be possible to restore values of variables after a call which did not change them. An alternate technique is to mark a subset of the variables as "read only."

The process of abstracting away part of the state and then re-attaching it is formalized in the following definition:

Definition 4. *Let L be parameterized by $(D,\phi,\otimes)$. Given a set of pairs $\Pi\subseteq L\times L$, the* parameterized extension *of Π w.r.t. ϕ (written $[\Pi]_\phi$) is defined as follows:*

$$[\Pi]_\phi\, w = \bigsqcap\{\phi(v',d)\mid (v,v')\in\Pi, d\in D, w\sqsubseteq\phi(v,d)\}$$

The parameterized extension is monotonic and conservatively approximates any function that is monotonic, homomorphic w.r.t. ϕ, and for which the input-output pairs are valid . (Proof omitted.)

In performing interprocedural analysis, we find appropriate sets of pairs iteratively by using one generation of sets of pairs as an approximation for calculating the next generation. One approach is to choose a finite subset of input states (call it A), set $\Pi_i^0 = A\times\{\bot\}$, then iteratively update via $\Pi_i^{j+1} = \{(v,\mathscr{F}_i(f_1,\ldots,f_n)(v))|(v,w)\in\Pi_i^j\}$. A much better approach would be to update only those input states which have been used at some actual call site. Even more efficient is to start with $\Pi_i^0=\emptyset$, and add to Π_i^{j+1} pairs of the form $(a,\bot)$ where a is any state in A that approximates the program state at a call site and is not yet an input state in Π_i^j. This strategy produces the same final results as the pointwise approximation for all states that actually appear at call sites, but not for the unused states.

Within this framework, the exact nature of an interprocedural analysis depends on several choices: the form of the abstract state and the set D, the $\otimes$ and ϕ functions, and the set A. The functors $\mathscr{F}_i$ are determined by procedure control flow. The ϕ function is the key to deciding which parts of the state are relevant. The set A determines how an infinite program state lattice is made finite; simple definitions of A result in more basic analyses while more careful definitions allow more interesting analyses.

3 Verifying JVML0

Having introduced the framework, we now use it to analyze JVML subroutines. In particular, we examine JVML0, a subset of JVML specifically designed to isolate subroutines. We allow recursive programs.

3.1 JVML0

JVML0 was created by Stata and Abadi [12] to help reason about JVML subroutines. A JVML0 program is a sequence of instructions, a set of local variables, and an operand stack. The language JVML0 has nine instructions. Stata and Abadi describe the semantics of these instructions fully.

$$\begin{aligned} \textit{instruction} ::= {} & \texttt{push}\ 0 \mid \texttt{inc} \mid \texttt{pop} \mid \texttt{store}\ x \mid \texttt{load}\ x \\ & \mid \texttt{jsr}\ L \mid \texttt{ret}\ x \mid \texttt{if}\ L \mid \texttt{halt} \end{aligned}$$

3.2 An Analysis

As mentioned previously, to define an analysis within our framework, we require a program state lattice L; parameters D, ϕ, and $\otimes$; and a finite set A representing the domain of input states for the sets of pairs.

A basic structure for the program state lattice follows immediately from the dynamic execution state. We define three types of program values: integers, whose type is i, references, whose type is a, and return address types, of the form $\langle L \rangle$ where L is the address of the subroutine (as in [11]). These types are incomparable; we add top ($\top$) and bottom ($\bot$) elements to form a lattice. Value types extend nicely to types for an operand stack S and a store F which maps local variables to values. The store is only a partial function; it is not defined for all variables. We also include a set C of return address types, ordered by $\subseteq$ in the program state. This set is used to track returns to subroutines other than the initial caller. We extend these lattices to (S,F,C) tuples and add top and bottom elements to form L.

Our domain D is a pair (S,F). We define ϕ and $\otimes$ as

$$\begin{aligned} \phi((S,F,C),(S',F')) &= (S \cdot S', F + F', C) \\ (S,F) \otimes (S',F') &= (S \cdot S', F + F') \end{aligned}$$

$$\texttt{ret}\ x^\sharp((S,F,C)) = \begin{cases} (S,\{(x,\top)\}+F,\{F[x]\}) & \exists L, F[x]=(L) \\ \top & \nexists L, F[x]=(L) \end{cases}$$

$$\texttt{halt}^\sharp((S,F,C)) = (S,F,\{\langle 0\rangle\})$$

$$\texttt{jsr}_1\ L^\sharp((S,F,C)) = \begin{cases} (\bar{S},\bar{F},\emptyset) & \langle L\rangle \in \bar{C} \\ \bot & \langle L\rangle \notin \bar{C} \end{cases}$$

$$\texttt{jsr}_2\ L^\sharp((S,F,C)) = \begin{cases} (\bar{S},\bar{F},\bar{C}-\{\langle L\rangle\}) & \bar{C}-\{\langle L\rangle\} \neq \emptyset \text{ and } \langle L\rangle \notin S \\ & \quad \text{and } \nexists v, (v,\langle L\rangle) \in F \\ (\bar{S},\bar{F},\bar{C}) & \langle L\rangle \in S \text{ or } \exists v, (v,\langle L\rangle) \in F \\ \bot & \bar{C}-\{\langle L\rangle\} = \emptyset \text{ and } \langle L\rangle \notin S \\ & \quad \text{and } \nexists v, (v,\langle L\rangle) \in F \end{cases}$$

$$\text{where } (\bar{S},\bar{F},\bar{C}) = [\Pi_L]_\phi(\langle\langle L\rangle\rangle \cdot S, F, C)$$

Fig. 1. Abstractions of `ret` and `jsr`

where $F+F'$ is as defined in Example 2.

Although subroutines are not marked by syntax, any instruction targeted by a `jsr` begins a subroutine and bears an annotation. All statements reachable from the starting instruction prior to a `ret` or `halt` instruction are considered to be part of the subroutine. This means that some statements are considered to belong to more than one subroutine. The first instruction begins a special main subroutine, and receives an initial annotation $\{(I,\bot)\}$ where I is the initial state. The endpoints for this subroutine are `halt` instructions (and `jsr` instructions as described below).

The functor for a subroutine starting at label L will be denoted $\mathscr{F}_L$. Its value is the join of the program state values at all end points of L. Most subroutine end points are those `ret` and `halt` instructions reachable from the beginning of the subroutine. However, because it is possible for a called subroutine to return to somewhere other than the current subroutine, `jsr` instructions also can be end points for the functor. But if all `jsr` instructions are treated as ordinary endpoints, the functor loses a great deal of precision. Therefore, we use the set C to include a `jsr` as an endpoint only if C indicates the (potential) existence of a return to a previous caller.

The abstractions for most instructions are clear; they must be strict over $\top$ and $\bot$. Figure 1 gives the abstractions for subroutine call and return (excepting top and bottom which, as mentioned, are strict). There are two abstractions for `jsr` because it acts differently as a subroutine endpoint ($\texttt{jsr}_2$) than it does as an instruction in the middle of a subroutine ($\texttt{jsr}_1$).

The $\texttt{jsr}_2\ L$ instruction is interesting. It needs to check two conditions: whether the set of returned-from subroutines contains only the callee and whether the address of the callee is present in the state. The former indicates that the callee is not returning to a (non-recursive) prior caller, while the latter indicates that the call is potentially recursive. A non-recursive call that only returns from L will never force this `jsr` to be an endpoint,

```
0  push0          5  push0
1  store 0        6  inc
2  jsr 5          7  push0
3  push0          8  if 10
4  halt           9  jsr 5
                 10  halt
```

Fig. 2. A simple JVML0 program

so $\texttt{jsr}_2$ returns bottom. A non-recursive call that returns from subroutines other than L may be returning to a prior caller, so the `jsr` is treated as an endpoint, but L is removed as a returned-from function. A potentially recursive call may be returning to the current caller or to a prior caller, so it always has a value. Recursive calls require that L stay in the set C, as we may be returning to a prior caller of L.

Finally, we choose our finite set A. We would like the analysis of a subroutine to ignore variables unused by that subroutine. The variables used by a subroutine can be calculated in an initial pass (similar to Freund and Mitchell [3], but not exactly the same, as we define subroutine boundaries differently). For each subroutine L, we can define the set V_L to be those variables which are modified in L or some subroutine called, directly or indirectly, from L. We can then define the finite set A_L as follows:

$$A_L = \{(S,F,C) | |S| < max \text{ and } DomF = V_L\}$$

Here *max* is some finite bound on the size of the partial stacks allowed in A_L. This bound is not an overall maximum bound on allowed stack height; rather, it is used to ignore portions of the stack which are unreachable by the subroutine. Ideally, *max* will have different values depending on the stack operations performed by the called subroutine.

If all abstract instructions map erroneous states to $\top$ and are strict in $\top$, then a program verifies if the output for each input-output pair in all annotations is not $\top$ (assuming a fixed point has been reached).

3.3 An Example

Example 5. Consider the JVML0 program in Figure 2. While this code is unlikely to be produced by a compiler, it allows us to demonstrate most of the interesting parts of the previous analysis.

The program has two subroutines, 0 and 5. Subroutine 5 is recursive. Subroutine 0 uses variable 0 while subroutine 5 uses no variables, so $V_0 = \{0\}$ and $V_5 = \emptyset$. The functors for this program are in Figure 3. Assuming *max* is set to 1, the iterations are given for the pairs of input and output states in Figure 4. The restriction of stack height to 1 in the pairs is safe because neither subroutine ever pops more than one item off the stack. This

$$
\begin{aligned}
\mathscr{F}_0 &= \mathtt{push0}^\sharp \circ \mathtt{store}^\sharp\ 1 \circ \mathtt{jsr}_1^\sharp\ 5 \circ \mathtt{push0}^\sharp \circ \mathtt{halt}^\sharp \\
&\quad \sqcup\ \mathtt{push0}^\sharp \circ \mathtt{store}^\sharp\ 1 \circ \mathtt{jsr}_2^\sharp\ 5 \\
\mathscr{F}_5 &= \mathtt{push0}^\sharp \circ \mathtt{inc}^\sharp \circ \mathtt{push0}^\sharp \circ \mathtt{if}^\sharp \circ \mathtt{jsr}_1^\sharp\ 5 \circ \mathtt{halt}^\sharp \\
&\quad \sqcup\ \mathtt{push0}^\sharp \circ \mathtt{inc}^\sharp \circ \mathtt{push0}^\sharp \circ \mathtt{if}^\sharp \circ \mathtt{jsr}_2^\sharp\ 5 \\
&\quad \sqcup\ \mathtt{push0}^\sharp \circ \mathtt{inc}^\sharp \circ \mathtt{push0}^\sharp \circ \mathtt{if}^\sharp \circ \mathtt{halt}^\sharp
\end{aligned}
$$

Fig. 3. Functors for program in Figure 2

$$
\begin{aligned}
\Pi_0^0 &= \{((\langle\rangle, \{(0,\top)\}, \emptyset), \bot)\} & \Pi_5^0 &= \{\} \\
\Pi_0^1 &= \{((\langle\rangle, \{(0,\top)\}, \emptyset), \bot)\} & \Pi_5^1 &= \{((\langle\langle 5\rangle\rangle, \{\}, \emptyset), \bot)\} \\
\Pi_0^2 &= \{(\langle\rangle, \{(0,\top)\}, \emptyset), \bot)\} & \Pi_5^2 &= \{((\langle\langle 5\rangle\rangle, \{\}, \emptyset), (\langle \mathbf{i}\rangle, \{\}, \{\langle 0\rangle\}))\} \\
\Pi_0^3 &= \{((\langle\rangle, \{(0,\top)\}\emptyset), (\langle \mathbf{i}\rangle, \{(0,\mathbf{i})\}, \{\langle 0\rangle\}))\} & \Pi_5^3 &= \{((\langle\langle 5\rangle\rangle, \{\}, \emptyset), (\langle \mathbf{i}\rangle, \{\}, \{\langle 0\rangle\}))\} \\
\Pi_0{}^* &= \{((\langle\rangle, \{(0,\top)\}, \emptyset), (\langle \mathbf{i}\rangle, \{(0,\mathbf{i})\}, \{\langle 0\rangle\}))\} & \Pi_5{}^* &= \{((\langle\langle 5\rangle\rangle, \{\}, \emptyset), (\langle \mathbf{i}\rangle, \{\}, \{\langle 0\rangle\}))\}
\end{aligned}
$$

Fig. 4. Iterations used when analyzing the program in Figure 2

speeds the iteration by grouping several possible operand stacks together for subroutine 5.

Notice how the pair $((\langle\langle 5\rangle\rangle, \{\}, \emptyset), \bot)$ is added upon the second iteration. This is in response to control in subroutine 0 reaching the $\mathtt{jsr}^\sharp\ 5$ instruction with state $(\langle\langle 5\rangle\rangle, \{(0,\mathbf{i})\}, \emptyset)$. The store is empty because its domain must equal V_5. This is safe as

$$\phi((\langle\langle 5\rangle\rangle, \{\}, \emptyset), (\langle\rangle, \{(0,\mathbf{i})\})) = (\langle\langle 5\rangle\rangle, \{(0,\mathbf{i})\}, \emptyset)$$

The `push0` on line 3 never has an effect on the analysis. This owes to the `jsr` instruction before it always receiving a value of $\{\langle 0\rangle\}$ in C, and therefore always passing bottom to the `push0`.

This example also illustrates a difficulty. We accept this program even though it contains an infinite recursion that will overflow the operand stack. We allow this by not enforcing an absolute maximum on stack height. We could add enforcement by utilizing a more complex representation for the operand stack that would include an approximation of the number of elements not explicitly represented.

The previous example, while short, does show how recursion and different kinds of returns are handled. It demonstrates how the parameterization can preserve the values of untouched state through a call. It also shows how a very small set of input-output pairs can characterize the behavior of a function on an infinite lattice (since the operand stack is not bounded).

The JVML (and especially JVML0) is an interesting and simple environment for investigating a fledgling interprocedural analysis framework. The experience of defining parameterizations for JVML0 will prove valuable when designing parameterizations for more complex situations, such as full JVML or full Java with method calls, which is our ultimate goal.

4 Related Work

Stata and Abadi [12] use JVML0 to consider subroutines in isolation. They provide both a semantics for JVML0 and a type system, which they formally prove sound. Programs type if each instruction satisfies constraints on the abstract program state before and after the instruction. The type system is not generative: the rules validate typings but do not provide them.

Hagiya and Tozawa [5] expand on Stata and Abadi, adding types for returns to distant callers and special types making variables ignored by subroutines polymorphic. They provide a dataflow algorithm that backtracks to search for the most appropriate types to assign variables.

Freund and Mitchell extend the work of Stata and Abadi to include object initialization [4], arbitrary returns and exception handlers. They formalize a framework [3] encompassing these, and have implemented a verifier using a three-phase algorithm which calculates subroutine membership and variable use before performing a dataflow analysis to determine types.

Qian [11] has a slightly different formalization of a large subset of JVML that also includes subroutines. Program states include subroutine records to track histories of variable modification. Statements are typed to a set of constraints. Qian provides an algorithm to compute solutions by repeated substitution.

O'Callahan [10] employs a type system based on STAL [7]. Unknown types are represented with type variables. Continuations are used for subroutine returns. Using a type variable for the (entire) stack permits a limited form of recursion.

Yelland [13] has formulated JVML bytecode in Haskell. He includes subroutines as calls to the (composition of) Haskell function(s) at the call site. These subroutines are similar to our functors, but use the actual subroutine at a call and not an approximation. As such, this approach requires a structured call stack and the absence of recursion.

As part of his survey of bytecode verification [6], Leroy presents two algorithms for verification across subroutines. The second is a polyvariate flow analysis that does not explicitly separate subroutines, and performs no extraction of relevant context.

Coglio [1] presents an analysis in which abstract values are sets of complete program states. This permits each program point to be represented as many states. Subroutine calls and returns are treated as special branches in control flow. The sets of complete states allow different states for all variables sent to a subroutine, modified or not.

5 Conclusion

While we have yet to demonstrate any capacity that has not been performed elsewhere, we believe that our approach is feasible and may easily be extended to a fuller subset of the language. Additionally, applying our framework successfully to JVML suggests that it could be successfully applied in other contexts.

References

[1] Alessandro Coglio. Simple verification technique for complex Java bytecode subroutines. Technical report, Kestrel Institute, December 2001.

[2] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, Los Angeles, California, USA, pages 238–252. ACM Press, New York, January 1977.

[3] Stephen N. Freund and John C. Mitchell. A formal framework for the Java bytecode language and verifier. *ACM SIGPLAN Notices*, 34(10):147–166, 1999.

[4] Stephen N. Freund and John C. Mitchell. The type system for object initialization in the Java bytecode language. *ACM Transactions on Programming Languages and Systems*, 21(6):1196–1250, 1999.

[5] Masami Hagiya and Akihiko Tozawa. On a new method for data flow analysis of Java Virtual Machine subroutines. In *Proc. 5th Static Analysis Symposium (SAS'98)*, volume 1503 of *Lecture Notes in Computer Science*, pages 17–32. Springer, September 1998.

[6] Xavier Leroy. Java bytecode verification: An overview. In *Computer Aided Verification*, pages 265–285. 2001.

[7] J. Gregory Morrisett, Karl Crary, Neal Glew, and David Walker. Stack-based typed assembly language. In *Types in Compilation*, pages 28–52. 1998.

[8] Alan Mycroft. The theory and practice of transforming call-by-need into call-by-value. In Bernard Robinet, editor, *Proceedings of the 4th International Symposium on Programming*, Paris, France, April 22–24, volume 83 of *Lecture Notes in Computer Science*, pages 269–281. Springer, Berlin, Heidelberg, New York, April 1980.

[9] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer, Berlin, Heidelberg, New York, 1999.

[10] Robert O'Callahan. A simple, comprehensive type system for java bytecode subroutines. In *Twenty-Sixth ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, Texas*, pages 70–78. January 1999.

[11] Zhenyu Qian. Standard fixpoint iteration for java bytecode verification. *Programming Languages and Systems*, 22(4):638–672, 2000.

[12] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 149–160. New York, NY, 1998.

[13] Phillip Yelland. A compositional account of the Java Virtual Machine. In *Proc. 26th ACM Symposium on Principles of Programming Languages (POPL'99)*, pages 57–69. January 1999.