Standard Fixpoint Iteration for Java Bytecode Verification

ZHENYU QIAN
Kestrel Institute

Java bytecode verification forms the basis for Java-based Internet security and needs a rigorous description. One important aspect of bytecode verification is to check if a Java Virtual Machine (JVM) program is statically well-typed. So far, several formal specifications have been proposed to define what the static well-typedness means. This paper takes a step further and presents a chaotic fixpoint iteration, which represents a family of fixpoint computation strategies to compute a least type for each JVM program within a finite number of iteration steps. Since a transfer function in the iteration is not monotone, we choose to follow the example of a nonstandard fixpoint theorem, which requires that all transfer functions are increasing, and monotone in case the bigger element is already a fixpoint. The resulting least type is the artificial top element if and only if the JVM program is not statically well-typed. The iteration is standard and close to Sun's informal specification and most commercial bytecode verifiers.

Categories and Subject Descriptors: D.3.1 [**Programming Languages**]: Formal Definitions and Theory

General Terms: Algorithms, Languages

Additional Key Words and Phrases: Java, bytecode verification, dataflow analysis, fixpoint

1. INTRODUCTION

The Java Virtual Machine (JVM) is a stack-based abstract computing machine. Java methods are usually compiled into JVM methods, which consist of JVM instructions. During execution, an (*operand*) *stack* and a set of registers called *local variables* are created on each method invocation and destroyed when the method execution completes.

In this paper, stack entries and local variables are uniformly called *memory locations.* They hold data that are either object references or values of primitive types. Object references point to objects stored in the *heap.* We consider only object references but no objects themselves, nor the heap in this paper.

Since a JVM method may be dynamically loaded from the network, there is no guarantee that it contains no bugs or has no hostile intentions to break the host system. Sun's JVM Specification [Lindholm and Yellin 1996] (SJVMS) requires, that prior to execution, *bytecode verification* must be performed to prove, among

The author was supported in part by DARPA contracts F30602-96-C-0363 and F30602-99-C-0091.
Author's address: Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA 94304.

other things, that each newly loaded JVM method is (statically) well-typed. Informally, if one can statically assign types to all memory locations at all program points[1] in a JVM method such that instructions only use and store data from and into memory locations with correct assigned types, then the method is *well-typed* and the assignment of types is a *legal* assignment for the method.

SJVMS informally describes the well-typedness of JVM programs. Since it is an important aspect of Java-based Internet security, a number of formal specifications have been proposed to define the well-typedness (e.g., Freund and Mitchell [1998], Goldberg [1998], Hagiya and Tozawa [1998], O'Callahan [1999b], [Pusch 1999], Qian [1998], and Stata and Abadi [1998]).

This paper takes a step further and presents a standard chaotic (fixpoint) iteration (e.g., see Cousot and Cousot [1979]), which represents a family of standard fixpoint computation strategies (e.g., see Kildall [1973] and Muchnick [1997]) to compute a least type for each JVM program within a finite number of iteration steps. Since the iteration is close to SJVMS and most commercial bytecode verifiers, it might be used to derive reference implementation. In addition, it might serve as a part in a comprehensive and relatively realistic formal model for Java-based Internet. Finally, it definitely contributes to the understanding of the JVM, in particular, the JVM subroutines.

The transfer functions in our chaotic iteration are defined based on a set of typing rules derived from those in the formal specification by Qian [1998]. One advantage is that the proofs of properties (with respect to the formal specification) become simpler than otherwise.

Unfortunately, we cannot directly apply any of the standard fixpoint theorems in the proofs, since not all transfer functions of the iteration are monotone (see Section 4.1), as required by these theorems (cf. Cousot and Cousot [1979] and Lassez et al. [1982] for a survey). The nonmonotonicity property stems from the requirements on instructions for JVM subroutines. To avoid the problem, we choose to follow the example of a nonstandard fixpoint theorem, which requires that all transfer functions are increasing, and monotone in case the bigger element is a fixpoint. The proof that all transfer functions satisfy the restricted monotonicity condition is not trivial, since as we will see later, some transfer functions do not always compute the sharpest type information from a local perspective.

Our chaotic iteration always yields a least type for each JVM program within a finite number of iteration steps. The least type is the artificial top element if and only if the JVM program is not statically well-typed with respect to the typing rules, upon which the transfer functions are defined.

In this paper we make a number of simplifying assumptions to JVM programs and bytecode verification. We believe that one can always discard the assumptions and extend the formalism to full bytecode verification for the entire JVM without affecting the main results of this paper.

The paper is organized as follows. Section 2 first recalls chaotic iteration and other related concepts, as well as a corresponding standard fixpoint theorem. Then it proves a nonstandard fixpoint theorem. Section 3 introduces JVM instructions

[1]Note that different types can be assigned to the same memory location at different program points.

and informally discusses assignments of types to them. Section 4 illustrates a nonmonotone transfer function, and informally explains why the transfer function is monotone in case where the bigger element is a fixpoint. Section 5 formally introduces some notations, in particular, defines the types for memory locations. The typing rules of the formal specification are presented in Section 6. Our chaotic iteration is described in Section 7. The formal properties are proved in Section 9. Section 10 discusses related work. Section 11 concludes the paper.

2. CHAOTIC ITERATION AND FIXPOINT THEOREMS

A *partially ordered set* (poset) $\langle D, \sqsubseteq\rangle$ consists of a set D and a partial order $\sqsubseteq$ on it. We use $d \sqsubset d'$ to denote $d \sqsubseteq d'$ and $d \neq d'$. A sequence $d_1, d_2, \cdots$ of elements in D is called *increasing* or simply a *chain* if $d_i \sqsubseteq d_{i+1}$ for all $i = 1, 2, \cdots$. It is called *strictly increasing* if $d_i \sqsubset d_{i+1}$ for all $i = 1, 2, \cdots$. The poset is of *finite height* if it contains no infinite, strictly increasing chain.

A function $f : D \to D$ is called *monotone* if for all $d, d' \in D$, $d \sqsubseteq d'$ implies that $f(d) \sqsubseteq f(d')$. Two functions $f, g : D \to D$ are said to *commute* if $f(g(d)) = g(f(d))$ for all $d \in D$. An element $d \in D$ is called a *fixpoint* of a function $f : D \to D$ if $f(d) = d$. A fixpoint $d \in D$ is called the *least* fixpoint if $d \sqsubseteq d'$ for all fixpoints $d' \in D$.

Consider a finite[2] set $\mathcal{F}$ of functions of the form $f : D \to D$. An element $d \in D$ is called a (*common*) *fixpoint* of $\mathcal{F}$ if it is a fixpoint of every function in the set.

Let $\langle D, \sqsubseteq\rangle$ be a poset with a bottom element $\perp$ (i.e., $\perp \sqsubseteq d$ for all $d \in D$). Let $\mathcal{F}$ be a finite set of monotone functions. A *chaotic iteration* produces infinite sequences of the following form

$$d_0, d_1, d_2, \cdots$$

where $d_0 = \perp$ and $d_i = g_i(d_{i-1})$ with $g_i \in \mathcal{F}$ for $i = 1, 2, \cdots$. The functions in $\mathcal{F}$ are called the *transfer functions* of the chaotic iteration.

The above chaotic iteration is called *fair* if whenever $f(d_i) \neq d_i$ for some finite $i \geq 0$ and some function $f \in \mathcal{F}$, $d_i \neq d_{i+k}$ for some finite $k \geq 1$. Intuitively, if progress is possible, then a fair chaotic iteration never applies the same function infinitely many times before making progress.

A special case of a standard fixpoint theorem by Cousot and Cousot [1979] is stated as follows.

THEOREM 2.1. *Let $\langle D, \sqsubseteq\rangle$ be a poset of finite height and with a bottom $\perp$. Let $\mathcal{F}$ be a finite set of monotone functions that pairwise commute. Then the least fixpoint of $\mathcal{F}$ exists in D. In particular, a fair chaotic iteration with the transfer functions in $\mathcal{F}$ can always produce the least fixpoint within a finite number of iteration steps.*

PROOF. First, the iteration produces only chains, since the bottom $\perp$ is the least element, and since the functions in $\mathcal{F}$ are monotone and pairwise commute.

Second, each produced chain contains a fixpoint d_h of $\mathcal{F}$ for a finite number h, since the iteration is fair and since the poset is of finite height.

Finally, the d_h found as above is the least fixpoint, since the bottom $\perp$ is the least element and that all functions in $\mathcal{F}$ are monotone. □

[2]The discussion here applies also for a countable set $\mathcal{F}$.

Unfortunately, as we will see later, the above fixpoint theorem is too restrictive to be applied here, since some transfer functions which arise here are not monotone. Thus we introduce a nonstandard fixpoint theorem.

A function $f : D \to D$ is called *increasing* if $d \sqsubseteq f(d)$ for all $d \in D$. Monotonicity does not imply increasingness. Consider a set consisting of two elements a, b such that $a \sqsubseteq b$. A function f satisfying $f(a) = f(b) = a$ is monotone but not increasing. The converse does not hold either. Consider another set with three elements a, b, c such that $a \sqsubseteq b \sqsubseteq c$. A function f satisfying $f(a) = c$, $f(b) = b$, and $f(c) = c$ is increasing but not monotone.

A function $f : D \to D$ is called *monotone-with-fixpoint* if for all $d, d' \in D$, when $d \sqsubseteq d'$ and d' is a fixpoint, then $f(d) \sqsubseteq d'$. A monotone function is obviously monotone-with-fixpoint, but the converse does not necessarily hold. Consider a set with three elements a, b, c and a function f such that $a \sqsubseteq b \sqsubseteq c$, $f(a) = b$, $f(b) = b$, and $f(c) = a$. The element b is the only fixpoint of f. The function f is monotone-with-fixpoint but not monotone.

By requiring that transfer functions be increasing and monotone-with-fixpoint rather than pairwise commuting and monotone, we obtain a nonstandard fixpoint theorem, whose proof is even more direct than that of Theorem 2.1.

THEOREM 2.2. *Let $\langle D, \sqsubseteq \rangle$ be a poset of finite height and with a bottom $\perp$. Let $\mathcal{F}$ be a finite set of functions that are increasing and monotone-with-fixpoint. Then the least fixpoint of $\mathcal{F}$ always exists in the poset. In particular, a fair chaotic iteration with the transfer functions in $\mathcal{F}$ can always reach the least fixpoint within a finite number of iteration steps.*

PROOF. First, by increasingness, the iteration produces only chains.

Second, using the same proof as for Theorem 2.1, each produced chain reaches a fixpoint d_h of $\mathcal{F}$ for a finite number h.

Finally, since $\perp$ is the least element, by monotonicity-with-fixpoint, the d_h found as above is the least fixpoint. $\square$

Our proof of the concrete chaotic iteration in this paper will follow the example of the proof of Theorem 2.2. For convenience, we will perform the proof directly, instead of applying Theorem 2.2. Note that in Theorem 2.2 it actually suffices to require that elements in chains produced by chaotic iteration starting with $\perp$ be increasing and monotone-with-fixpoint. Thus we will require monotonicity-with-fixpoint on these elements only.

3. BYTECODE

We consider only classes but no interfaces. Furthermore, we consider only one primitive type `int`, among many. We will use `i` to denote type `int`.

We consider only methods that return no results (i.e., whose return type is `void`). A *method body* is a sequence of instructions. The position of an instruction is called an *address*.

The execution of a method always starts with a newly created (operand) stack and n newly created local variables for a number n specified in the method. Initially, the stack is empty, and some local variables contain the `this` object and all actual arguments.

We only consider the following instructions in method bodies.

```
Method void m(i)  // Signature, where the argument of the method is of type i.
.limit locals 4   // The method has 4 local variables. Store this object into
                  // local variable 1 and the actual argument into local variable 2.
                  // Create an empty stack.
 0 jsr 5          // Push address 1 onto the stack and jump to address 5.
 1 aload 1        // Copy the object in local variable 1 onto the stack
 2 astore 2       // Move the top object from the stack into local variable 2
 3 jsr 5          // Push address 4 onto the stack and jump to address 5.
 4 return         // Return.
 5 astore 3       // Move the return address from the stack into local variable 3.
 6 jsr 7          // Push address 7 onto the stack and jump to address 7.
 7 astore 4       // Move the return address from the stack into local variable 4.
 8 ret 3          // Return to the address stored in local variable 3.
```

Fig. 1. A JVM program.

—`aload` x: pushes the object or address in local variable x onto the stack.

—`iload` x: pushes the value of type `i` in local variable x onto the stack.

—`astore` x: pops the object or address at the top of the stack from the stack and stores it into local variable x.

—`istore` x: pops the value of type `i` at the top of the stack from the stack and stores it into local variable x.

—`ifnull` p: jumps to address p if the top stack element is *null*.

—`return`: terminates the current method.

—`jsr` p: pushes the next address onto the stack and jumps to address p.

—`ret` x: jumps to an address stored in local variable x.

`jsr` and `ret` instructions can be used together to realize *subroutines*, which typically implement `finally` clauses in `Java` programs. Concretely, one can use a `jsr` p instruction to push the next address as the return address onto the stack and to jump to *subroutine* p. In subroutine p, one can use an `astore` x instruction to store the return address from the stack into a local variable x. Finally, one can use a `ret` x instruction to return from subroutine p to the return address stored in local variable x. In general, a subroutine need not have a `ret` instruction, and a `ret` instruction may return to any indirect caller in the subroutine call stack.

We allow multiple `ret` instructions for each subroutine, whereas SJVMS requires that each subroutine have at most one `ret` instruction.

Figure 1 shows a JVM method and describes its meaning. Note that the instruction `jsr 5` at addresses 0 and 3 calls subroutine 5, and the instruction `astore 3` at address 5 stores the return address (1 or 4) into local variable 3. Subroutine 5 contains the instruction `jsr 7` at address 6, which calls subroutine 7. Subroutine 7 has the instruction `ret 3` at address 8, which completes both subroutines 7 and 5, and returns to the return address (i.e., 1 or 4) as stored in local variable 3.

3.1 Types for Memory Locations

Figure 2 shows a legal assignment of memory types to memory locations in the method in Figure 1. For the moment we will not consider how to compute such an assignment. In general, a method can have zero or more legal assignments.

Code	VT	ST	SR	Succ
`Method void m(i)`				
`.limit locals 4`				
`0 jsr 5`	[c,i,u,u]	[]	[]	5
`1 aload 1`	[c,i,u,u]	[]	[]	2
`2 astore 2`	[c,i,u,u]	[c]	[]	3
`3 jsr 5`	[c,c,u,u]	[]	[]	5
`4 return`	[c,c,u,u]	[]	[]	return
`5 astore 3`	[c,u,u,u]	$[\langle 5\rangle]$	$[(5,\{\})]$	6
`6 jsr 7`	[c,u,$\langle 5\rangle$,u]	[]	$[(5,\{3\})]$	7
`7 astore 4`	[c,u,$\langle 5\rangle$,u]	$[\langle 7\rangle]$	$[(5,\{3\}),(7,\{\})]$	8
`8 ret 3`	[c,u,$\langle 5\rangle$,$\langle 7\rangle$]	[]	$[(5,\{3\}),(7,\{4\})]$	1,4

Fig. 2. Types for the method in Figure 1.

In the figure, we write types for local variables under column VT and types for stack entries under column ST. Local variables are numbered with 1, 2, 3, and 4 from left to right. Intuitively, the assigned types at an instruction indicate the types of data the memory locations may hold prior to the execution of that instruction.

The assignment also contains extensions of subroutine call stacks under column SR. We will discuss them later in this section.

Column Succ is not part of the assignment. It contains static successors of each address. It is helpful in the illustration, since the assigned types at an address are often related to those at static successors.

A type for a memory location is either a class (name), or the primitive type `i`, or a term of the form $\langle p\rangle$, where p is an address, or a special symbol `u`.

Type `u` is a type of all possible data. Bytecode verification treats each memory location with type `u` as *unusable*, since no single instruction can deal with all possible data in such a memory location in a type-correct way.[3]

Within a given method, a type of the form $\langle p\rangle$ is a type of all addresses a subroutine starting with the address p may return to. We consider return address types because memory locations may hold return addresses. Two return address types are distinct and represent disjoint sets of return addresses if and only if the associated subroutine addresses are distinct.

We assume that the method in Figure 2 is declared in a class with name `c`. The types are `c`, `i`, $\langle 5\rangle$, $\langle 7\rangle$, and `u`. In particular, type $\langle 5\rangle$ is the type of the values 1 and 4, i.e., the return addresses that subroutine 5 may return to, and type $\langle 7\rangle$ is the type of the value 7, i.e., the return address that subroutine 7 may return to.

We define a partial order $\sqsubseteq$ on all types for memory locations by that $t \sqsubseteq t'$ if and only if at least one of the following holds:

(1) $t = t'$.
(2) $t' = \mathtt{u}$.
(3) Both t and t' are classes, and t is a (direct or indirect) subclass of t'.

For the types given for the method above we have that $\mathtt{c} \sqsubseteq \mathtt{u}$, $\mathtt{i} \sqsubseteq \mathtt{u}$, and $\langle p\rangle \sqsubseteq \mathtt{u}$ for $p = 5$ and 7. These types build a semilattice with respect to $\sqsubseteq$.

[3]In the full JVM, some stack manipulation instructions like `pop` can handle memory locations with type `u`. But we do not consider them in this paper.

To check that the assigned types do indicate the types of data the memory locations may hold prior to the execution of an instruction, we see that for example, local variables 1 and 2 at address 0, which are assigned with types `c` and `i`, respectively, will hold the `this` object and the actual argument, respectively, at run time. Local variables 3 and 4 at address 0, which are assigned with type `u`, will contain undefined data. The local variables at address 5 are assigned with types bigger than or equal to those both at addresses 0 and 3; the stack at address 5 is assigned with type $[\langle 5 \rangle]$. This coincides with the execution of the instruction `jsr 5` at addresses 0 and 3, which pushes their return addresses onto the stack and jumps to address 5.

The process of assigning types to memory locations for the instruction `ret 3` at address 8 is complex due to special requirements for a `ret` instruction. The first requirement is that the static successors of the instruction should depend on the type assigned to the given local variable. For the instruction `ret 3` at address 8, since the type for local variable 3 is $\langle 5 \rangle$, the successors are all return addresses for subroutine 5, i.e., addresses 1 and 4.

The second requirement is that none of the return addresses for a subroutine and all its inner subroutines should be usable outside the subroutine. To achieve this, we assign type `u` to the memory locations containing such return addresses. In the above example, types $\langle 5 \rangle$ and $\langle 7 \rangle$ for local variables 3 and 4 at address 8 are replaced by type `u` at addresses 1 and 4.

The third requirement is that if a local variable is not modified in a subroutine, then its content at each calling site of the subroutine should remain usable after the call completes. To ensure this, we do the following things:

—If a local variable is not modified in a subroutine, then its type at each return address should be bigger than or equal to that at the corresponding calling site, independent of that at any `ret` instruction of the subroutine.

—Otherwise, its type at each return address should be bigger than or equal to that at each `ret` instruction of the subroutine, independent of that at the corresponding calling site.

In the example, local variable 2 is not modified in subroutine 5. Thus the type of local variable 2 at the return address 1 is chosen to be type `i` at the calling site 0, and the type at the return address 4 is chosen to be type `c` at the calling site 3.

These requirements imply that we need to record called subroutines and modified local variables. To do this, we introduce a concept of *subroutine records*. In Figure 2, subroutine records are given under column *SR*. A subroutine record is a list of pairs where the first elements in these pairs form a subroutine call stack, and the second element in each pair is a set of local variables that are directly modified in the corresponding subroutine. Using the subroutine record at a `ret` instruction, we can compute the set of all local variables modified (directly or indirectly) in a subroutine.

In the above example, the subroutine record $[(5, \{3\}), (7, \{4\})]$ at address 8 records that subroutines 5 and 7 are called, and that local variable 3 is directly modified in subroutine 5, and local variable 4 in subroutine 7. The set of all local variables modified in subroutine 5 is the union $\{3\} \cup \{4\}$. In other words, we know that local variables 1 and 2 are not modified in subroutine 5.

```
Method void n(i)  // The argument of the method is of type i.
.limit locals 3   // The method has 3 local variables. Store this object into
                  // local variable 1 and the actual argument into local variable 2.
                  // Create an empty stack.
 0 aload 1        // Copy the object in local variable 1 onto the stack.
 1 ifnull 4       // If the top stack element is null then jump to address 4.
 2 aload 1        // Copy the object in local variable 1 onto the stack again.
 3 astore 2       // Move the top object from the stack into local variable 2.
 4 jsr 6          // Push address 5 onto the stack and jump to address 6.
 5 return         // Return.
 6 astore 3       // Move the top object from the stack into local variable 3.
 7 aload 1        // Copy the object in local variable 1 onto the stack.
 8 ifnull 11      // If the top stack element is null then jump to 11.
 9 iload 2        // Copy the integer in local variable 2 onto the stack.
 10 istore 2      // Move the top integer from the stack into local variable 2.
 11 ret 3         // Return to the address stored in local variable 3.
```

Fig. 3. Another example

Note that only those local variables that are modified by an `astore` or `istore` instruction are required to be put into the subroutine record. This is slightly different from SJVMS, which requires that all variables accessed by an `aload`, `iload`, or `ret` instruction be put into the subroutine records as well.

In this paper, a subroutine record is designed to be a list, which is close to the current implementation in JDK 1.2. For a more general treatment, where a subroutine record is a directed acyclic graph, see Qian [1998].

4. THE PROBLEM AND EXPLANATION OF OUR SOLUTION

In this section, we show why the transfer function for a `ret` instruction is not monotone, and briefly explain our solution. We will use the method in Figure 3 for illustration and assume that the method is declared in class `c`.

We first informally illustrate how to use dataflow analysis to assign types and subroutine records for the entire method.

Let us start with the definition of a relation $\sqsubseteq$ on tuples of memory location types and subroutine records for individual addresses, by lifting componentwise the relation $\sqsubseteq$ on the types defined in the previous section and the subset relation on sets of modified local variables. We introduce an artificial bottom element $\perp$ as the least element into the set of these tuples.

Then we define a relation $\sqsubseteq$ on all assignments of memory location types and subroutine records for the entire method by componentwise lifting the relation $\sqsubseteq$ on the tuples just introduced above. The least assignment is

$$\{p \mapsto \perp \mid \text{for each address } p \text{ in the method}\}.$$

Figure 4 shows the first 10 iteration steps of our dataflow analysis on the example in Figure 3, each computing a new assignment of memory location types and subroutine records for the entire method. Column "Step" indicates these analysis steps at specified addresses. The analysis starts with the least assignment. Since each analysis step computes a new assignment by updating the previously assigned types and subroutine records at all successors of the specified address (as given

			ϕ_1		
Step	Code	Succ	VT	ST	SR
1	`Method void n(i)`	0			
	`.limit locals 3`				
2	`0 aload 1`	1	$[\mathtt{c},\mathtt{i},\mathtt{u}]$	$[]$	$[]$
3	`1 ifnull 4`	2,4	$[\mathtt{c},\mathtt{i},\mathtt{u}]$	$[\mathtt{c}]$	$[]$
9	`2 aload 1`	3	$[\mathtt{c},\mathtt{i},\mathtt{u}]$	$[]$	$[]$
10	`3 astore 2`	4	$[\mathtt{c},\mathtt{i},\mathtt{u}]$	$[\mathtt{c}]$	$[]$
4	`4 jsr 6`	6	$[\mathtt{c},\mathtt{i},\mathtt{u}]$	$[]$	$[]$
			$[\mathtt{c},\mathtt{u},\mathtt{u}]$	$[]$	$[]$
	`5 return`			$\perp$	
5	`6 astore 3`	7	$[\mathtt{c},\mathtt{i},\mathtt{u}]$	$[\langle 6\rangle]$	$[(6,\{\})]$
6	`7 aload 1`	8	$[\mathtt{c},\mathtt{i},\langle 6\rangle]$	$[]$	$[(6,\{3\})]$
7	`8 ifnull 11`	9,11	$[\mathtt{c},\mathtt{i},\langle 6\rangle]$	$[\mathtt{c}]$	$[(6,\{3\})]$
8	`9 iload 2`	10	$[\mathtt{c},\mathtt{i},\langle 6\rangle]$	$[]$	$[(6,\{3\})]$
	`10 istore 2`	11	$[\mathtt{c},\mathtt{i},\langle 6\rangle]$	$[\mathtt{i}]$	$[(6,\{3\})]$
	`11 ret 3`	5	$[\mathtt{c},\mathtt{i},\langle 6\rangle]$	$[]$	$[(6,\{3\})]$

Fig. 4. An assignment for the method in Figure 3

in column "Succ"), it is enough for Figure 4 to show only the updates at these successors for each analysis step. In fact, the updates are the least upper bounds of the newly computed types and subroutine records and the previously assigned, corresponding ones at all successor addresses. For notational simplicity, Figure 4 also omits the bottom element $\perp$ in the initial assignment for all addresses except address 5; the bottom element $\perp$ at address 5 is not updated within the first 10 iteration steps.

Let us consider the analysis steps in detail. Step 1 produces an assignment at address 0 based on the information in the method head and the operational semantics for the execution, i.e., on the facts that at address 0, local variable 1 holds the `this` object, local variable 2 the actual parameter, local variable 3 an undefined value, and both the stack and the subroutine record are empty.

Step 2 at address 0 produces an assignment at the successor address 1 using that at address 0, since the stack at address 1 holds `this` object.

The `ifnull` instruction at address 1 has successor addresses 2 and 4. Thus step 3 updates the assignments at addresses 2 and 4.

As step 4, the chaotic iteration chooses to continue at address 4. Since the instruction `jsr 6` pushes the next address 5 onto the stack and the address 5 is of type $\langle 6\rangle$ for subroutine 6, step 4 produces the stack containing a single type $\langle 6\rangle$ and the subroutine record containing an initial assignment $[(6,\{\})]$ at address 6. The empty set in the subroutine record means that so far no local variables have been modified in subroutine 6.

Since the instruction `astore 3` at address 6 moves the top element of the stack into local variable 3, step 5 at address 6 produces a subroutine record at address 7 recording that local variable 3 is modified in subroutine 6.

The iteration proceeds in a similar way as above. Note that the instruction `ifnull 11` at address 8 has static successor addresses 9 and 11. Thus step 7 produces an assignment at each successor address.

After step 8, the iteration chooses to continue at address 2 and then at address 3. Since the instruction at address 3 is `astore 2`, local variable 2 at address 4 must

Code	ϕ_2		
	VT	ST	SR
...	$\cdots$ same as ϕ_1 $\cdots$		
`10 istore 2`	$[c, \mathtt{i}, \langle 6\rangle]$	$[\mathtt{i}]$	$[(6, \{3\})]$
`11 ret 3`	$[c, \mathtt{i}, \langle 6\rangle]$	$[]$	$[(6, \{2, 3\})]$

Fig. 5. Another assignment for the method in Figure 3

	ϕ'_1			ϕ'_2		
Code	VT	ST	SR	VT	ST	SR
...	$\cdots$ same as in ϕ_1 $\cdots$			$\cdots$ same as in ϕ_2 $\cdots$		
`4 jsr 6`	$[\mathtt{c}, \mathtt{u}, \mathtt{u}]$	$[]$	$[]$	$[\mathtt{c}, \mathtt{u}, \mathtt{u}]$	$[]$	$[]$
`5 return`	$[\mathtt{c}, \mathtt{u}, \mathtt{u}]$	$[]$	$[]$	$[\mathtt{c}, \mathtt{i}, \mathtt{u}]$	$[]$	$[]$
...	$\cdots$ same as in ϕ_1 $\cdots$			$\cdots$ same as in ϕ_2 $\cdots$		
`11 ret 3`	$[\mathtt{c}, \mathtt{i}, \langle 6\rangle]$	$[]$	$[(6, \{3\})]$	$[\mathtt{c}, \mathtt{i}, \langle 6\rangle]$	$[]$	$[(6, \{2, 3\})]$

Fig. 6. A nonmonotone transfer function for the method in Figure 3

now hold both an integer and an object, which have no common type except type `u`. Thus step 10 assigns type `u` to local variable 2 at address 4.

4.1 A Nonmonotone Transfer Function

Write ϕ_1 for the final assignment obtained in Figure 4. Now our chaotic iteration may choose to continue at address 10. Since the instruction at address 10 is `istore 2`, this step causes the addition of local variable 2 in the set of "modified" local variables and hence produces the assignment at address 11 as shown in Figure 5, where we write ϕ_2 for the resulting assignment. Clearly $\phi_1 \sqsubseteq \phi_2$ holds.

Now we consider two continuations of the iteration at address 11, one from assignment ϕ_1, the other from ϕ_2. Figure 6 illustrates the results ϕ'_1 and ϕ'_2. Since assignment ϕ_1 does not record, that at address 11, local variable 2 is modified in subroutine 6, the result ϕ'_1 has type `u` for local variable 2 at address 5, which comes from that at address 4. Since assignment ϕ_2 records that at address 11, local variable 2 is modified in subroutine 6, the result ϕ'_2 has type `i` for local variable 2 at address 5, which comes from that at address 11.

Since $\phi_1 \sqsubseteq \phi_2$ and $\phi'_1 \not\sqsubseteq \phi'_2$, the transfer function for a `ret` instruction is nonmonotone!

It is clear that the nonmonotonicity property stems from the fact that addition of a new local variable in the set of modified local variables may cause re-computation of a type for this local variable. The problem lies in the fact that the type obtained in the re-computation comes from a different place than the previous type for this local variable.

4.2 An Intuitive Explanation of Our Solution

The problem with nonmonotonicity is not an unsolved problem. One option is to design a new algorithm where nonmonotonicity disappears. Another is to prove that the given algorithm as a whole still works in spite of nonmonotonicity.

Following the first approach, one might design an algorithm with two phases. The first phase would compute modified local variables only. The second would assign types based on the sets of modified local variables. Both phases need to examine almost the whole program because in order to compute a complete set of

Code	$VT(2)$ in ϕ_3	$VT(2)$ in ϕ'_3	$VT(2)$ in ϕ'
...			
4 jsr 6	t_1		t'_1
5 return	t_3	$lub(t_1, t_2)$	t'_3
...			
11 ret 3	t_2		t'_2

Fig. 7. An intuitive explanation of our solution

all modified local variables, one needs to know almost all static execution paths. Thus one disadvantage of the approach would be that the resulting algorithm may be inefficient. Another disadvantage would be that it would differ significantly from most commercial bytecode verifiers and thus not directly provide hints on their qualities. Freund and Mitchell's recent work [Freund and Mitchell 1999b] follows this direction. See Section 10 for more discussion.

This paper follows the second approach. It has the disadvantage that the proofs are nonstandard. But it does have the advantage that the chaotic iteration is simple, natural, and close to SJVMS and most commercial bytecode verifiers.

Now we informally explain our proof, following the example of the proof of Theorem 2.2. The proof of increasingness is easy. The nontrivial task is to prove monotonicity-with-fixpoint. The complication lies in the treatment of the `jsr` and `ret` instructions.

Let us use the method in Figure 3 again and consider the types for local variable 2 at addresses `4`, `5`, and `11` as illustrated in Figure 7. Assume that the assignment ϕ_3 does not record local variable 2 as a modified local variable at address 11. Assume also that the iteration produces an assignment ϕ'_3 from ϕ_3 in a similar way as it produces ϕ'_2 in Figure 6 from ϕ_1 in Figure 4, i.e., first adds local variable 2 as a modified local variable and then computes a new type $lub(t_1, t_2)$ for local variable 2 at address 5. Finally, assume that the assignment ϕ' is a legal assignment of the method and satisfies $\phi_3 \sqsubseteq \phi'$. Next, we explain why $\phi'_3 \sqsubseteq \phi'$.

Note that although computing the least upper bound $lub(t_1, t_2)$ is standard in a dataflow analysis, it does not compute the sharpest type information here, since the type t_2, not type $lub(t_1, t_2)$, is the sharpest type information. Choosing $lub(t_1, t_2)$ ensures the increasingness property. But we need to prove that it is not too big, i.e., $lub(t_1, t_2) \sqsubseteq t'_3$.

Since $\phi_3 \sqsubseteq \phi'$, $t_1 \sqsubseteq t'_1$ and $t_2 \sqsubseteq t'_2$. In order to show that $lub(t_1, t_2) \sqsubseteq t'_3$, we need only to show that $lub(t'_1, t'_2) \sqsubseteq t'_3$, i.e., $t'_1 \sqsubseteq t'_3$ and $t'_2 \sqsubseteq t'_3$. The proof of $t'_2 \sqsubseteq t'_3$ is straightforward, since ϕ' surely records that local variable 2 is a modified local variable at address 11. The proof of $t'_1 \sqsubseteq t'_3$ is due to the existence of a static execution path from address 4 to address 11 on which local variable 2 is not modified. Such a path does exist; otherwise, the assignment ϕ_3, which does not record that local variable 2 is a modified local variable at address 11, would not have been produced. Actually, looking at Figure 3, we see that local variable 2 is not modified on the path of addresses 4, 6, 7, 8, and 11.

Since the assignment ϕ' is legal for the method in consideration, applying chaotic iteration will not change it. In particular, if transfer functions are applied at each program point one by one on the path, the assignment ϕ' will not change. This means that the type of local variable 2 in ϕ' along the path either increases or remains unchanged. This means that $t'_1 \sqsubseteq t'_2$. Recall $t'_2 \sqsubseteq t'_3$. Hence $t'_1 \sqsubseteq t'_3$.

5. PRELIMINARIES

In the rest of the paper, we use the notation $\overline{\alpha_n}$ to denote a sequence $\alpha_1, \cdots, \alpha_n$, and the notation $\{\cdots\}$ to denote a set. We use $\{\overline{\alpha_n \mapsto \alpha'_n}\}$, where $\alpha_i \neq \alpha_j$ holds for all $0 \leq i \neq j \leq n$, to denote a (finite) mapping. We define $\mathcal{D}om(\{\overline{\alpha_n \mapsto \alpha'_n}\}) \stackrel{def}{=} \{\overline{\alpha_n}\}$. For a mapping θ and an element $\alpha \in \mathcal{D}om(\theta)$, we use $\theta(\alpha)$ to denote the result of the mapping for α. Note that the expression $\theta(\alpha)$ represents an application of a predefined "apply" function to two (first-order) terms θ and α. We use $\theta[\alpha \mapsto \alpha']$ to denote the mapping that is equal to θ except it maps α to α'. For a set $\mathcal{W}$, we use $\theta_{|\mathcal{W}}$ to denote the mapping obtained from θ by restricting its domain to $\mathcal{D}om(\theta) \cap \mathcal{W}$. A *list* $[\overline{\alpha_n}]$ is a special mapping $\{\overline{n \mapsto \alpha_n}\}$. We define $size([\overline{\alpha_n}]) \stackrel{def}{=} n$ and $[\overline{\alpha_n}] + \alpha \stackrel{def}{=} [\overline{\alpha_n}, \alpha]$.

In formalizing our specification, we chose to use a constraint-solving framework based on a first-order order-sorted algebra, which consists of a collection of sets, called *sorts*, a subset relation among the sorts, *functions*, and *predicates* on these sorts (cf. Smolka et al. [1989]). A function is uniquely determined by a name. A *predicate* is uniquely determined by a name and argument sorts.

There is a set of *variables* for each sort. Variables may occur in terms and logical formulas and are placeholders for terms. One should not confuse variables in the algebra with local variables in JVM programs.

Terms are built by functions and variables and are well-sorted. A term of a sort is automatically a term of each of its supersorts. Each term has a least sort.

Logical formulas are built as in first-order predicate logic, using predicates, terms, usual logical constants, connectives, and quantifiers, and are well-sorted. A set of logical formulas represents the conjunction of them; the empty set represents $true$.

We use $\mathcal{FV}(r)$ to denote the set of all free variables in a term or a logical formula r. A term or a logical formula r is *closed* if $\mathcal{FV}(r) = \emptyset$.

A closed logical formula semantically represents a statement of a relation in the first-order order-sorted algebra, which evaluates either to $true$ or $false$.

For simplicity, we omit the explicit definitions of sorts as well as those of standard functions and predicates in this paper.

A *substitution* is a mapping $\{\overline{X_n \mapsto r_n}\}$, where each X_i is a variable and each r_i a closed term of the sort of X_i for all $i = 1, \cdots, n$. We use σ to range over all substitutions. Applying a substitution σ to a term or logical formula r yields a result $\sigma(r)$ defined as usual, where all free occurrences of $X \in \mathcal{D}om(\sigma)$ in r are replaced by $\sigma(X)$, and bound variables in $\sigma(r)$ are automatically renamed to avoid bound variable capture.

A *constraint* is a logical formula. A constraint r is *satisfied under* a substitution σ if and only if $\sigma(r)$ is closed and $\sigma(r)$ evaluates to $true$ (in the first-order order-sorted algebra).

5.1 Methods and Types

For the rest of the paper, we assume an arbitrary but fixed environment consisting of a fixed and finite set of classes and a fixed subclass relation. We use c to range over all these classes.

We consider an arbitrary but fixed method described as $(\mathtt{c}, \overline{\mathtt{t_m}}, \mathtt{n}, \mathtt{mth})$. The notation $\mathtt{c}$ stands for the name of the class in which the method is declared, $\mathtt{m}$ for the number of the arguments of the method, and $\overline{\mathtt{t_m}}$ for the types of the arguments of the method, where each $\mathtt{t}_i$ for $1 \leq i \leq \mathtt{m}$ is either a class or primitive type $\mathtt{i}$. The notation $\mathtt{n}$ stands for the number of local variables in the method with $\mathtt{m} \leq \mathtt{n}$. The notation $\mathtt{mth}$ stands for a (possibly empty) list of instructions as the *method body*.

We use p and s to range over all addresses of the method body $\mathtt{mth}$, i.e., $0 \leq p, s < \mathit{size}(\mathtt{mth})$, where s usually stands for (the beginning of) a subroutine. We use x to range over all indices of local variables, i.e., $1 \leq x \leq \mathtt{n}$.

Types for memory locations are of the following forms:

$$t ::= c \mid \mathtt{i} \mid \langle p \rangle \mid \mathtt{u}$$

A partial order $\sqsubseteq$ is defined by that $t \sqsubseteq t'$ if and only if at least one of the following holds:

(1) $t = t'$;

(2) $t' = \mathtt{u}$;

(3) $t = c$, $t' = c'$, and c is a (direct or indirect) subclass of c'.

We define a join operation $\sqcup$ based on $\sqsubseteq$ in the usual way. All types form a join-semilattice.

It is worth emphasizing that the join-semilattice is of finite height, since the subclass relation in a finite set of classes is always of finite height.

We construct five sorts of composite types in the following.

Type lists:	vt, st	$::= [\overline{t_n}]\ (n \geq 0)$
Index sets:	xs	$::= \{\overline{x_n}\}\ (n \geq 0)$
Subroutine records:	sr	$::= [\overline{(s_n, xs_n)}]\ (n \geq 0, s_i \neq s_j \text{ for } i \neq j)$
Program point types (P-types):	ρ	$::= (vt, st, sr) \mid \bot_P \mid \top_P$
Method types (M-types):	ϕ	$::= \{p \mapsto \rho_p \mid 0 \leq p < \mathit{size}(\mathtt{mth})\} \mid \top_M$

We usually use vt for types of local variables and st for types of entries in a stack.

A P-type is an assignment of types and subroutine records at an individual program point. An M-type is a complete assignment of types and subroutine records for an entire method.

We define a partial order $\sqsubseteq$ on lists of types and a partial order $\sqsubseteq$ on subroutine records as follows:

—$[\overline{t_n}] \sqsubseteq [\overline{t'_m}]$ if and only if $n = m$ and $t_i \sqsubseteq t'_i$ for all $i = 1, \cdots, n$.

—$[\overline{(s_n, xs_n)}] \sqsubseteq [\overline{(s'_m, xs'_m)}]$ if and only if $n = m$, $s_i = s'_i$, and $xs_i \subseteq xs'_i$ for all $i = 1, \cdots, n$.

We define a partial order $\sqsubseteq$ on P-types by that $\rho \sqsubseteq \rho'$ if and only if at least one of the following conditions holds:

—$\rho = \bot_P$;

—$\rho' = \top_P$;

—$\rho = (vt, st, sr) \ \wedge\ \rho' = (vt', st', sr') \ \wedge\ vt \sqsubseteq vt' \ \wedge\ st \sqsubseteq st' \ \wedge\ sr \sqsubseteq sr'$ for some vt, st, sr, vt', st', and sr'.

We define a join-operation $\sqcup$ on P-types based on the $\sqsubseteq$ in the standard way. All P-types form a join-semilattice of finite height.

We define a partial order $\sqsubseteq$ on M-types by that $\phi \sqsubseteq \phi'$ if and only if at least one of the following conditions holds:

—$\phi' = \top_M$;

—$\phi \neq \top_M \ \wedge \ \phi' \neq \top_M \ \wedge \ (\forall p.\phi(p) \sqsubseteq \phi'(p))$

The element $\top_M$ is introduced for easy formulation of chaotic iteration. By definition, we have that

$$\top_M \neq \{p \mapsto \top_P \mid \text{ for each } p\}.$$

We introduce an abbreviation:

$$\bot_M \stackrel{def}{=} \{p \mapsto \bot_P \mid \text{ for each } p\}$$

We define a join-operation $\sqcup$ on M-types based on the $\sqsubseteq$ in the standard way. All M-types form a join-semilattice of finite height.

For convenience, we extend the partial orders defined above on types and composite types to a partial order $\sqsubseteq$ on substitutions as follows: $\sigma_1 \sqsubseteq \sigma_2$ if and only if $\mathcal{D}om(\sigma_1) = \mathcal{D}om(\sigma_2)$, and for each $X \in \mathcal{D}om(\sigma_1)$ the following holds:

$$\begin{array}{ll} \sigma_1(X) \sqsubseteq \sigma_2(X) & \text{if the sort of } X \text{ is the sort of types or} \\ & \text{one of the five sorts of composite types} \\ \sigma_1(X) = \sigma_2(X) & \text{otherwise} \end{array}$$

6. TYPING RULES

6.1 The Form of Typing Rules

Our typing rules use the variables in the following table.

Variables	P, S	VT, ST	X	C	Y	SR	Φ
For elements	p, s	vt, st	x	c	c and $\langle p \rangle$	sr	ϕ

Note that we have omitted the formal definition of sorts and the subsort relation for simplicity. Actually we should have defined, for example, that the sort of variable Y is a supersort of that of variable C, and the sort of variables VT and ST is a list sort whose element sort is a supersort of the sort of variable Y.

A typing rule R is always of the form

$$\begin{array}{c} \mathcal{A}_R \\ \hline \mathcal{C}_R \\ \mathcal{S}_R \end{array}$$

where the hypothesis part (above the line) is a set $\mathcal{A}_R$ of logical formulas, and the conclusion part (under the line) consists of two sets $\mathcal{C}_R$ and $\mathcal{S}_R$ of logical formulas. Intuitively, the set $\mathcal{A}_R$ determines an address of application. The set $\mathcal{C}_R$ constrains the P-type at the determined address. The set $\mathcal{S}_R$ constrains the P-types at each successor of the determined address.

In the description of each concrete typing rule R in Section 6.2, the determination of the sets $\mathcal{C}_R$ and $\mathcal{S}_R$ is always implicit: the set $\mathcal{C}_R$ consists of all those logical formulas in the conclusion part that are *not* of the form

$$\cdots \sqsupseteq \cdots,$$

while the set $\mathcal{S}_R$ consists of all those that are of the form.

Definition 6.1. For each typing rule R, we use the notations

$$\begin{aligned} Q_R &\stackrel{def}{=} \mathcal{FV}(\mathcal{A}_R) - \{\Phi\} \\ Q'_R &\stackrel{def}{=} \mathcal{FV}(\mathcal{C}_R \cup \mathcal{S}_R) - (\{\Phi\} \cup Q_R). \end{aligned}$$

Rule R *induces* a constraint

$$\forall Q_R.(\mathcal{A}_R \Rightarrow \exists Q'_R.(\mathcal{C}_R \cup \mathcal{S}_R)).$$

As we will see later, Φ is a common variable occurring in all typing rules; it stands for an M-type of the method body `mth`. Actually Φ is the only free variable in the constraints induced by typing rules.

Note that although the sets $\mathcal{C}_R$ and $\mathcal{S}_R$ play the same role in a constraint induced by a typing rule, they will play different roles in the definition of transfer functions.

All typing rules together define a set of constraints on variable Φ, i.e., on M-types of the method body `mth`.

Definition 6.2. An M-type ϕ is called *legal* if and only if $\phi \neq \top_M$ and the constraint set

$$\{\forall Q_R.(\mathcal{A}_R \Rightarrow \exists Q'_R.(\mathcal{C}_R \cup \mathcal{S}_R)) \mid \text{for each typing rule } R\}$$

is satisfied under the substitution $\{\Phi \mapsto \phi\}$.

A method body may have zero or more legal M-types.

Definition 6.3. A method body is called *well-typed* if it has a legal M-type.

6.2 Concrete Typing Rules

This section gives concrete typing rules. In order to discuss the instances of the typing rules, we use the method in Figure 2, which is declared in class `c`, has a single argument type `i` and has 4 local variables.

We assume the following in the description of the rules:

—In each typing rule R except rule (1), the set $\mathcal{A}_R$ always implicitly contains the condition $\Phi(P) \neq \bot_P$.

—In rules (2), (3), (4), (5), and (6), the set $\mathcal{C}_R$ always implicitly contains the condition $P + 1 < \mathit{size}(\mathtt{mth})$.

Figure 8 gives the first group of typing rules. Rule (1) induces constraints on P-types at address 0 based the information contained in the header of the method. Note that $\mathcal{C}_{(1)} = \emptyset$. The P-type at address 0 in Figure 2 satisfies rule (1), where $\mathtt{m} = 1$, $\mathtt{t}_1 = \mathtt{i}$, and $\mathtt{n} = 4$.

Rule (2) is straightforward except one point: the condition $VT(X) = Y$ implicitly forces the local variable X to have a type of the form c or $\langle p \rangle$, since Y can be

$$\frac{0 < size(\mathtt{mth})}{\Phi(0) \sqsupseteq ([1 \mapsto c_0, 2 \mapsto \mathtt{t}_1, \cdots, \mathtt{m}+1 \mapsto \mathtt{t}_\mathtt{m}, \mathtt{m}+2 \mapsto \mathtt{u}, \cdots, \mathtt{n} \mapsto \mathtt{u}], [], [])} \tag{1}$$

$$\frac{\mathtt{mth}(P) = \mathtt{aload}\ X}{\begin{array}{l}\Phi(P) = (VT, ST, SR)\\ VT(X) = Y\\ \Phi(P+1) \sqsupseteq (VT, ST+Y, SR)\end{array}} \tag{2}$$

$$\frac{\mathtt{mth}(P) = \mathtt{iload}\ X}{\begin{array}{l}\Phi(P) = (VT, ST, SR)\\ VT(X) = \mathtt{i}\\ \Phi(P+1) \sqsupseteq (VT, ST+\mathtt{i}, SR)\end{array}} \tag{3}$$

$$\frac{\mathtt{mth}(P) = \mathtt{astore}\ X}{\begin{array}{l}\Phi(P) = (VT, ST+Y, SR)\\ \Phi(P+1) \sqsupseteq (VT[X \mapsto Y], ST, ad\_mvs(\{X\}, SR))\end{array}} \tag{4}$$

$$\frac{\mathtt{mth}(P) = \mathtt{istore}\ X}{\begin{array}{l}\Phi(P) = (VT, ST+\mathtt{i}, SR)\\ \Phi(P+1) \sqsupseteq (VT[X \mapsto \mathtt{i}], ST, ad\_mvs(\{X\}, SR))\end{array}} \tag{5}$$

$$\frac{\mathtt{mth}(P) = \mathtt{ifnull}\ P'}{\begin{array}{l}\Phi(P) = (VT, ST+C, SR)\\ \Phi(P') \sqsupseteq (VT, ST, SR)\\ \Phi(P+1) \sqsupseteq (VT, ST, SR)\end{array}} \tag{6}$$

Fig. 8. The first group of typing rules.

instantiated only by these forms. As an example, rule (2) is satisfied at address 1 in Figure 2 under $\{P \mapsto 1, X \mapsto 1, Y \mapsto \mathtt{c}, VT \mapsto [\mathtt{c}, \mathtt{i}, \mathtt{u}, \mathtt{u}], ST \mapsto [], SR \mapsto []\}$. Rule (3) is similar to rule (2).

Rules (4) and (5) use the following auxiliary function to extend the set of modified local variables of the last subroutine in a nonempty subroutine record; the function returns an empty subroutine record if the input subroutine record is empty.

$$ad\_mvs(xs, [\overline{(s_n, xs_n)}]) \stackrel{def}{=} \begin{cases} [\overline{(s_{n-1}, xs_{n-1})}] + (s_n, xs_n \cup xs) & \text{if } n \geq 1 \\ [] & \text{if } n = 0 \end{cases}$$

In Figure 2, rule (4) is satisfied at address 5 under $\{P \mapsto 5, X \mapsto 3, VT \mapsto [\mathtt{c}, \mathtt{u}, \mathtt{u}, \mathtt{u}], ST \mapsto [\langle 5 \rangle], Y \mapsto \langle 5 \rangle, SR \mapsto [(5, \{\})]\}$, with $ad\_mvs(\{3\}, [(5, \{\})]) = [(5, \{3\})]$. Note that rules (4) and (5) are the only rules that require addition of modified variables.

Rule (6) is straightforward.

Note that no rule for a `return` instruction is necessary, since a `return` instruction introduces no explicit constraints.

Figure 9 contains the remaining typing rules. Rule (7) is for the `jsr` instruction. The auxiliary predicate *called* is defined by

$called(s, [\overline{(s_n, xs_n)}])$ yields true if and only if $s = s_i$ for some i with $1 \leq i \leq n$.

Intuitively, the condition $not(called(S, SR))$ ensures that no subroutines are called

$$\frac{\texttt{mth}(P) = \texttt{jsr}\ S}{\begin{array}{l}\Phi(P) = (VT, ST, SR)\\ not(called(S, SR))\\ \Phi(S) \sqsupseteq (VT, ST + \langle S\rangle, SR + (S, \{\}))\end{array}} \tag{7}$$

$$\frac{\texttt{mth}(P) = \texttt{ret}\ X}{\begin{array}{l}\Phi(P) = (VT, ST, SR)\\ VT(X) = \langle S\rangle\end{array}} \tag{8}$$

$$\frac{\begin{array}{l}\texttt{mth}(P) = \texttt{ret}\ X\\ \Phi(P) = (VT, ST, SR)\\ VT(X) = \langle S\rangle\\ \texttt{mth}(P') = \texttt{jsr}\ S\\ \Phi(P') = (VT', ST', SR')\\ called(S, SR)\end{array}}{\begin{array}{l}P' + 1 < size(\texttt{mth})\\ \Phi(P'+1) \sqsupseteq (VT'[j \mapsto filter(VT(j), sbs\_in(S,SR)) \mid j \in mvs\_in(S,SR)],\\ \qquad filter\_l(ST, sbs\_in(S,SR)), ad\_mvs(mvs\_in(S,SR), sr\_bef(S, SR))\end{array}} \tag{9}$$

Fig. 9. The second group of typing rules.

recursively. The condition

$$\Phi(S) \sqsupseteq (VT, ST + \langle S\rangle, SR + (S, \{\}))$$

says that at the beginning of subroutine S, the top element in the stack should be able to hold a return address. In addition, it adds an entry $(S, \{\})$ to the subroutine record. In Figure 2, rule (7) is satisfied both at address 0 under $\{P \mapsto 0, S \mapsto 5, SR \mapsto [], \cdots\}$, and at address 3 under $\{P \mapsto 3, S \mapsto 5, SR \mapsto [], \cdots\}$.

Rule (8) is for the `ret` instruction. It ensures that local variable X is of type $\langle S\rangle$ for subroutine S. Note that $\mathcal{S}_{(8)} = \emptyset$. In Figure 2, the M-type satisfies rule (8) at address 8 under $\{P \mapsto 8, X \mapsto 3, VT \mapsto [c, \mathtt{u}, \langle 5\rangle, \langle 7\rangle], \cdots\}$ with $VT(3) = \langle 5\rangle$.

In rule (9), the set $\mathcal{A}_{(9)}$ determines an address P, where a `ret` instruction occurs, and an address P', where a corresponding `jsr` instruction occurs. The predicate $called(S, SR)$ ensures that the applications of auxiliary functions in $\mathcal{S}_{(9)}$ are well-defined.

The set $\mathcal{S}_{(9)}$ in rule (9) uses several auxiliary functions. The auxiliary function ad_mvs has been defined before. Other auxiliary functions are defined as follows. Note that due to the existence of the predicate $called(S, SR)$ in the set $\mathcal{A}_{(9)}$, we need only to consider the definitions for a subroutine record $[\overline{(s_n, xs_n)}]$ and a subroutine s_k with $1 \le k \le n$.

$$\begin{array}{rcl} sbs\_in(s_k, [\overline{(s_n, xs_n)}]) & \stackrel{def}{=} & \{s_k, \cdots, s_n\}\\ mvs\_in(s_k, [\overline{(s_n, xs_n)}]) & \stackrel{def}{=} & xs_k \cup \cdots \cup xs_n\\ sr\_bef(s_k, [\overline{(s_n, xs_n)}]) & \stackrel{def}{=} & [\overline{(s_{k-1}, xs_{k-1})}] \end{array}$$

Intuitively, the function $sbs\_in(s_k, [\overline{(s_n, xs_n)}])$ computes the set of all subroutines that are called within subroutine s_k (including s_k) according to subroutine call record $[\overline{(s_n, xs_n)}]$. The function $mvs\_in(s_k, [\overline{(s_n, xs_n)}])$ computes the set of all variables that are modified in subroutine s_k or other inner subroutines according to subroutine call record $[\overline{(s_n, xs_n)}]$,. The function $sr\_bef(s_k, [\overline{(s_n, xs_n)}])$ computes the prefix of subroutine record $[\overline{(s_n, xs_n)}]$ before subroutine s_k (exclusive of s_k).

We define two auxiliary (filter) functions for an arbitrary type t, a type list $[\overline{t_m}]$, and a set ss of subroutines as follows:

$$\begin{aligned} \mathit{filter}(t, ss) &\stackrel{def}{=} \begin{cases} \mathtt{u} & \text{if } t = \langle s \rangle \text{ and } s \in ss \\ t & \text{otherwise} \end{cases} \\ \mathit{filter\_l}([\overline{t_m}], ss) &\stackrel{def}{=} [\overline{\mathit{filter}(t_m, ss)}] \end{aligned}$$

The function $\mathit{filter}(t, ss)$ changes type t into type $\mathtt{u}$ if it is a return address type for a subroutine in the set ss. The function $\mathit{filter\_l}([\overline{t_m}], ss)$ does the same for each element t_i in a type list $[\overline{t_m}]$. These two functions are used in rule (9) to ensure that some return addresses become unusable when a subroutine returns.

In rule (9), the term

$$VT'[j \mapsto \mathit{filter}(VT(j), sbs\_in(S,SR)) \mid j \in mvs\_in(S,SR)]$$

represents some types of the local variables, where if local variable j is modified in subroutine S according to subroutine call record SR, then its type is either $VT(j)$ or type $\mathtt{u}$, depending on whether $VT(j)$ is a return address type for a subroutine in $sbs\_in(S,SR)$ or not; otherwise its type is $VT'(j)$. The term

$$\mathit{filter\_l}(ST, sbs\_in(S,SR))$$

represents a type list obtained from type list ST, where all return address types for subroutines in $sbs\_in(S,SR)$ are changed into type $\mathtt{u}$. The term

$$ad\_mvs(mvs\_in(S,SR), sr\_bef(S, SR))$$

is a subroutine call record obtained from the prefix of subroutine call record SR before subroutine S, where all local variables modified within subroutine S are recorded as modified by the caller subroutine of subroutine S.

In Figure 2, rule (T-9) is satisfied at address 8 under $\{P \mapsto 8, X \mapsto 3, S \mapsto 5, P' \mapsto 0, SR \mapsto [(5,\{3\}),(7,\{4\})]\}$ (or $\{P \mapsto 8, X \mapsto 3, S \mapsto 5, P' \mapsto 3, SR \mapsto [(5,\{3\}),(7,\{4\})]\}$) with

$$sbs\_in(S,SR) = \{5,7\},\ mvs\_in(S,SR) = \{3,4\} \text{ and } sr\_bef(S,SR) = [].$$

At the successor $P' + 1 = 1$ (and $P' + 1 = 4$ as well), we have that

$$\begin{aligned} &\mathit{filter}(VT(j), sbs\_in(S,SR)) = \mathtt{u} \text{ for } j = 3 \text{ or } 4 \\ &\mathit{filter\_l}(ST, sbs\_in(S,SR)) = [] \\ &ad\_mvs(mvs\_in(S,SR), sr\_bef(S,SR)) = [] \end{aligned}$$

Note that our specification requires that only those local variables that are modified by `astore` or `istore` be put into subroutine records. The SJVMS requires that all local variables accessed by `astore`, `istore`, `aload`, `iload`, or `ret` be put into subroutine records.

7. CHAOTIC ITERATION FOR BYTECODE VERIFICATION

In this section, we present a chaotic iteration to compute legal M-types. The transfer functions of the iteration are derived from the typing rules in the previous section. For technical reasons, we will design a generic transfer function, which can be instantiated into a concrete transfer function for each typing rule. The generic transfer function takes a substitution σ as argument, which contains not only an M-type $\sigma(\Phi)$ for variable Φ, but in most cases also other information such as the address $\sigma(P)$ for variable P, indicating where the typing rule should be applied.

Definition 7.1. Consider a typing rule R and a substitution σ.

—Rule R is called *applicable* under σ if $\mathcal{D}om(\sigma) = \mathcal{FV}(\mathcal{A}_R) \cup \{\Phi\}$ and $\sigma(\mathcal{A}_R)$ holds.
—Rule R is called *pre-satisfied* under σ if $\mathcal{D}om(\sigma) = \mathcal{FV}(\mathcal{A}_R \cup \mathcal{C}_R) \cup \{\Phi\}$, and both $\sigma(\mathcal{A}_R)$ and $\sigma(\mathcal{C}_R)$ hold.
—Rule R is called *satisfied* under σ if it is pre-satisfied under σ and $\sigma(\mathcal{S}_R)$ holds.
—Rule R is said to *fail* under σ if it is applicable under σ but there are no substitutions σ' such that $\sigma'_{|\mathcal{FV}(\mathcal{A}_R)\cup\{\Phi\}} = \sigma$ and $\sigma'(\mathcal{C}_R)$ holds.

Intuitively, applicability indicates that a typing rule can be used at a given address. Pre-satisfaction indicates, that in addition to applicability, checking the P-type at the given address is successful. Satisfaction means, that in addition to pre-satisfaction, checking the P-types at all successor addresses of the given address is successful. The concept of failure captures the only reason for the chaotic iteration to fail.

Informally we distinguish between several situations:

—If a typing rule is not applicable, then it has no impact on the given input.
—The fact that a typing rule is pre-satisfied but not satisfied indicates that the input M-type is too small at some successor addresses; one should choose a bigger M-type.
—If a typing rule is applicable but not pre-satisfied, then there are no strictly bigger M-types that pre-satisfy the typing rule. In this case, the chaotic iteration fails.

The generic transfer function is defined as procedure $\mathcal{P}(\sigma, R)$ in Figure 10. The three steps of the procedure roughly capture the cases above. In particular, the second case corresponds to the production of the M-type

$$\sigma'(\Phi)[\sigma'(A) \mapsto (\sigma'(\Phi(A)) \sqcup \sigma'(B)) \mid (\Phi(A) \sqsupseteq B) \in \mathcal{S}_R]$$

which is the least M-type among those that are bigger than or equal to $\sigma(\Phi)$ and satisfy all predicates in $\mathcal{S}_R$. As we will see below, the caller of the procedure always secures that $\sigma(\Phi) \neq \top_M$. Note that $\mathcal{FV}(A) \cup \mathcal{FV}(B) \subseteq \mathcal{D}om(\sigma')$ and thus that the terms $\sigma'(A)$ and $\sigma'(B)$ are always defined, since we always have that

$$\mathcal{FV}(\mathcal{A}_R \cup \mathcal{C}_R) \cup \{\Phi\} \supseteq \mathcal{FV}(\mathcal{S}_R).$$

Figure 11 shows our chaotic iteration. It repeatedly calls procedure $\mathcal{P}(\sigma, R)$ to compute a sequence of *intermediate* M-types $\phi_0, \phi_2, \cdots$. As we will prove later, the iteration always terminates, finally yields as result the least legal M-type of the method body when the method body has any legal M-types, or finally yields M-type $\top_M$ when the method body has no legal M-types.

7.1 Examples

As an extreme case, assume that `mth` is an empty list. Then there are only two distinct M-types: the bottom element $\perp_M = \{\}$ and the top element $\top_M$. Use the

Procedure $\mathcal{P}(\sigma, R)$

(1) **if** rule R is not applicable under σ **then return** $\sigma(\Phi)$
(2) **else if** there is a substitution σ' with $\sigma'_{|\mathcal{FV}(\mathcal{A}_R)\cup\{\Phi\}} = \sigma$ such that rule R is pre-satisfied under σ' **then return** the M-type

$$\sigma'(\Phi)[\sigma'(A) \mapsto (\sigma'(\Phi(A)) \sqcup \sigma'(B)) \mid (\Phi(A) \sqsupseteq B) \in \mathcal{S}_R]$$

(3) **else return** M-type $\top_M$.

Fig. 10. Generic transfer function.

Iteration $\mathcal{I}$:

(1) $\phi_0 \stackrel{def}{=} \bot_M$; $k := 0$;
(2) **while** $\phi_k \neq \top_M$ and there exist a typing rule R and a substitution σ with $\mathcal{D}om(\sigma) = \mathcal{FV}(\mathcal{A}_R) \cup \{\Phi\}$, $\sigma(\Phi) = \phi_k$ and $\phi_k \neq \mathcal{P}(\sigma, R)$
do $\phi_{k+1} \stackrel{def}{=} \mathcal{P}(\sigma, R)$; $k := k + 1$ **od**;
(3) **return** ϕ_k

Fig. 11. Chaotic iteration.

notations in Definition 6.1. It is straightforward to see that for none of the typing rules R, the set $\mathcal{A}_R$ is satisfied under any substitution of variables in Q_R. Thus all constraints induced by the typing rules are trivially satisfied. Therefore, $\bot_M$ is a legal M-type. It is easy to see that iteration $\mathcal{I}$ yields $\bot_M$.

Assume that `mth` consists of only a `return` instruction (at address 0). Then it is easy to see that for none of the typing rules R except rule (1), the set $\mathcal{A}_R$ is satisfied under any substitution of variables in Q_R. Thus the constraints induced by all typing rules except rule (1) are trivially satisfied. Examining rule (1), it is not hard to see that two of the legal M-types are

$$\{0 \mapsto ([1 \mapsto c_0, 2 \mapsto \mathtt{t}_1, \cdots, \mathtt{m}+1 \mapsto \mathtt{t}_\mathtt{m}, \mathtt{m}+2 \mapsto \mathtt{u}, \cdots, \mathtt{n} \mapsto \mathtt{u}], [], [])$$
$$\{0 \mapsto \top_P\} \text{ (note that } \{0 \mapsto \top_P\} \neq \top_M).$$

Iteration $\mathcal{I}$ actually yields the first one.

Figure 12 describes an execution of iteration $\mathcal{I}$, which finally yields the legal M-type in Figure 2. Column "Step" contains numbers indicating the order of calls of transfer functions (i.e., $\mathcal{P}(\sigma, R)$); column "Rule" contains the typing rules corresponding to the called transfer functions; and column "Succ" contains the static successor addresses of each address. The iteration starts with the bottom M-type $\bot_M$. (The figure does not show $\bot_M$ explicitly.) Each row in the figure contains an updated P-type in an intermediate M-type in the produced sequence. A P-type may be updated several times. For simplicity, we only write those P-types that have changed from one M-type to the next one.

For example, step 1 uses a transfer function corresponding to rule (1) to compute (an M-type with) P-type $([\mathtt{c},\mathtt{i},\mathtt{u},\mathtt{u}], [], [])$ at address 0. Step 2 uses a transfer function corresponding to rule (7), checking the applicability and pre-satisfaction with respect to the previous P-type at address 0 and computes an update $([\mathtt{c},\mathtt{i},\mathtt{u},\mathtt{u}], [\langle 5\rangle], [])$ at address 5, and so on.

Recall that in each typing rule R except rule (1), the set $\mathcal{A}_R$ always implicitly contains the constraint $\Phi(P) \neq \bot$. So iteration $\mathcal{I}$ always starts with an application of the transfer function corresponding to rule (1), if the method body is non-empty.

Step	Rule	Code	Succ	VT	ST	SR
1	(1)	`Method void m(i)`	`0`			
		`.limit locals 4`				
2	(7)	`0 jsr 5`	`5`	[c,i,u,u]	[]	[]
8	(2)	`1 aload 1`	`2`	[c,i,u,u]	[]	[]
9	(4)	`2 astore 2`	`3`	[c,i,u,u]	[c]	[]
10	(7)	`3 jsr 5`	`5`	[c,c,u,u]	[]	[]
		`4 return`		[c,c,u,u]	[]	[]
3	(4)	`5 astore 3`	`6`	[c,i,u,u]	$[\langle 5\rangle]$	$[(5,\{\})]$
11	(4)		`6`	[c,u,u,u]		
4	(7)	`6 jsr 7`	`7`	$[$c,i,$\langle 5\rangle$,u$]$	[]	$[(5,\{3\})]$
12	(7)		`7`	$[$c,u,$\langle 5\rangle$,u$]$		
5	(4)	`7 astore 4`	`8`	$[$c,i,$\langle 5\rangle$,u$]$	$[\langle 7\rangle]$	$[(5,\{3\}),(7,\{\})]$
13	(4)		`8`	$[$c,u,$\langle 5\rangle$,u$]$		
6	(8)	`8 ret 3`		$[$c,i,$\langle 5\rangle,\langle 7\rangle]$	[]	$[(5,\{3\}),(7,\{4\})]$
7	(9)		`1`			
14	(8)			$[$c,u,$\langle 5\rangle,\langle 7\rangle]$		
15	(9)		`1`			
16	(9)		`4`			

Fig. 12. Compute the M-type in Figure 2.

It is worth noticing that steps 15 and 16 use a transfer function corresponding to rule (9) at address 8, where the subroutine record is $sr = [(5,\{3\}),(7,\{4\})]$, and thus $mvs\_in(5, sr) = \{3, 4\}$, $sbs\_in(5, sr) = \{5, 7\}$, and $sr\_bef(5, sr) = []$. Step 15 produces [c,i,u,u] for local variables at address 1, where local variables 3 and 4 have type u from those at address 8 because $3, 4 \in mvs\_in(5, sr)$ and $5, 7 \in sbs\_in(5, sr)$, and local variables 1 and 2 have types c and i from those at address 0 because $1, 2 \notin mvs\_in(5, sr)$. Note that local variable 2 at address 1 does not have type u from that at address 8. Step 16 produces [c,c,u,u] at address 4 based on the same principle, where local variable 2 has type c from that at address 3.

8. ITERATION $\mathcal{I}$ AND LEGAL M-TYPES

This section focuses on the relationship between iteration $\mathcal{I}$ and legal M-types. First of all, we formally establish that a legal M-type is always a fixpoint of procedure $\mathcal{P}$.

LEMMA 8.1. Consider an arbitrary typing rule R and a substitution σ with $\mathcal{D}om(\sigma) = \mathcal{FV}(\mathcal{A}_R) \cup \{\Phi\}$ such that $\sigma(\Phi)$ is a legal M-type. Then we have $\mathcal{P}(\sigma, R) = \sigma(\Phi)$.

PROOF. If rule R is not applicable under σ, then by the definition of procedure $\mathcal{P}$, $\mathcal{P}(\sigma, R) = \sigma(\Phi)$.

Now assume that rule R is applicable under σ. Thus $\sigma(\mathcal{A}_R)$ holds. Recall that $Q_R \stackrel{def}{=} \mathcal{FV}(\mathcal{A}_R) - \{\Phi\}$ and $Q'_R \stackrel{def}{=} \mathcal{FV}(\mathcal{C}_R \cup \mathcal{S}_R) - (\{\Phi\} \cup Q_R)$. Since $\sigma(\Phi)$ is a legal M-type, by definition, the constraint

$$\forall Q_R.(\mathcal{A}_R \Rightarrow \exists Q'_R.(\mathcal{C}_R \cup \mathcal{S}_R))$$

is satisfied under $\{\Phi \mapsto \sigma(\Phi)\}$. This means that there is a substitution σ' with $\mathcal{D}om(\sigma') = Q_R \cup Q'_R \cup \{\Phi\}$ such that $(\sigma')_{|\mathcal{FV}(\mathcal{A}_R)\cup\{\Phi\}} = \sigma$ and $\sigma'(\mathcal{C}_R)$ and $\sigma'(\mathcal{S}_R)$

hold. Then rule R is pre-satisfied under σ'. By the definition of procedure $\mathcal{P}$,

$$\mathcal{P}(\sigma', R) = \sigma'(\Phi)[\sigma'(A) \mapsto (\sigma'(\Phi(A)) \sqcup \sigma'(B)) \mid (\Phi(A) \sqsupseteq B) \in \mathcal{S}_R].$$

Since $\sigma'(\mathcal{S}_R)$ holds, $\sigma'(\Phi(A)) \sqsupseteq \sigma'(B)$, and thus $\sigma'(\Phi(A)) \sqcup \sigma'(B) = \sigma'(\Phi(A))$ hold for each $(\Phi(A) \sqsupseteq B) \in \mathcal{S}_R$. Hence $\mathcal{P}(\sigma, R) = \sigma'(\Phi) = \sigma(\Phi)$. $\square$

Now we formally state that iteration $\mathcal{I}$ is correct (with respect to the typing rules).

THEOREM 8.2 (CORRECTNESS). *If iteration $\mathcal{I}$ terminates with an M-type $\phi \neq \top_M$, then M-type ϕ is a legal M-type.*

PROOF. Consider an arbitrary typing rule R. Recall that $Q_R \stackrel{def}{=} \mathcal{FV}(\mathcal{A}_R) - \{\Phi\}$ and $Q'_R \stackrel{def}{=} \mathcal{FV}(\mathcal{C}_R \cup \mathcal{S}_R) - (\{\Phi\} \cup Q_R)$. We need only to prove that the constraint

$$\forall Q_R.(\mathcal{A}_R \Rightarrow \exists Q'_R.(\mathcal{C}_R \cup \mathcal{S}_R))$$

is satisfied under $\{\Phi \mapsto \phi\}$.

For doing this, assume that there is a substitution σ such that $\mathcal{D}om(\sigma) = \mathcal{FV}(\mathcal{A}_R) \cup \{\Phi\}$, $\sigma(\Phi) = \phi$ and such that $\sigma(\mathcal{A}_R)$ holds. Thus R is applicable under σ. Since ϕ is the result of iteration $\mathcal{I}$, $\phi = \mathcal{P}(\sigma, R)$. By the definition of procedure $\mathcal{P}$, since $\phi \neq \top_M$, there is a substitution σ' such that $\sigma'_{|\mathcal{FV}(\mathcal{A}_R) \cup \{\Phi\}} = \sigma$, and rule R is pre-satisfied under σ'. Thus $\mathcal{D}om(\sigma') = Q_R \cup Q'_R \cup \{\Phi\}$, $\sigma'(\mathcal{C}_R)$ holds and

$$\phi = \sigma'(\Phi)[\sigma'(A) \mapsto (\sigma'(\Phi(A)) \sqcup \sigma'(B)) \mid (\Phi(A) \sqsupseteq B) \in \mathcal{S}_R].$$

Let us choose an arbitrary $\Phi(A) \sqsupseteq B$ in $\mathcal{S}$. Then $\phi(\sigma'(A)) = \sigma'(\Phi(A)) \sqcup \sigma'(B)$. Since $\sigma'(\Phi) = \sigma(\Phi) = \phi$, $\sigma'(\Phi(A)) = \sigma'(\Phi(A)) \sqcup \sigma'(B)$. Thus $\sigma'(\Phi(A)) \sqsupseteq \sigma'(B)$. Since $\Phi(A) \sqsupseteq B$ is arbitrary in $\mathcal{S}$, $\sigma'(\mathcal{S}_R)$ holds. Since rule R and σ have been arbitrarily chosen, the assertion of the theorem holds. $\square$

9. TERMINATION OF ITERATION $\mathcal{I}$ AND LEAST M-TYPES

This section mainly focuses on the operational properties of the iteration. Recall that Theorem 2.2 requires the following conditions:

—The poset is of finite height and has a bottom element.

—All transfer functions are increasing.

—All transfer functions are monotone-with-fixpoint.

—The iteration is fair.

Let the join-semilattice of all M-types be the poset. The first condition trivially holds. The increasingness and fairness conditions are easy. They will be discussed in Section 9.1. Monotonicity-with-fixpoint is not that easy to prove. First, some preparations are made in Sections 9.2 and 9.3. Section 9.4 formally addresses monotonicity-with-fixpoint. Finally Section 9.5 contains the results on least M-types.

9.1 Increasingness and Termination

We first prove that procedure $\mathcal{P}$ is increasing.

LEMMA 9.1 (INCREASINGNESS). For each typing rule and each substitution σ with $\mathcal{D}om(\sigma) = \mathcal{FV}(\mathcal{A}_R) \cup \{\Phi\}$, $\sigma(\Phi) \sqsubseteq \mathcal{P}(\sigma, R)$.

PROOF. Let us use the notations in Figure 10. In the case when rule R is not applicable under σ, $\mathcal{P}(\sigma, R) = \sigma(\Phi)$

In the case when there is a substitution σ' with $\sigma'_{|\mathcal{FV}(\mathcal{A}_R)\cup\{\Phi\}} = \sigma$ such that rule R is pre-satisfied under σ', then

$$\mathcal{P}(\sigma, R) = \sigma'(\Phi)[\sigma'(A) \mapsto (\sigma'(\Phi(A)) \sqcup \sigma'(B)) \mid (\Phi(A) \sqsupseteq B) \in \mathcal{S}_R].$$

Since $\sigma'(\Phi(A)) \sqsubseteq \sigma'(\Phi(A)) \sqcup \sigma'(B)$, $\sigma(\Phi) = \sigma'(\Phi) \sqsubseteq \mathcal{P}(\sigma, R)$.

In other cases, $\sigma(\Phi) \sqsubset \top_M = \mathcal{P}(\sigma, R)$. □

Based on the increasingness property, we prove the termination of the chaotic iteration.

THEOREM 9.2 (TERMINATION). *Iteration $\mathcal{I}$ is terminating.*

PROOF. Consider any arbitrary sequence $\phi_0, \phi_1, \cdots$ of intermediate M-types produced by iteration $\mathcal{I}$. By Lemma 9.1, since $\phi_k \neq \phi_{k+1}$, the sequence is strictly increasing. Since the semilattice of M-types has finite height, the iteration process can only have finite number of iteration steps. It is easy to check that all auxiliary functions and predicates introduced in this paper are terminating. Thus iteration $\mathcal{I}$ is terminating. □

9.2 Some Basic Properties

We formalize the observation that if a typing rule is applicable or pre-satisfied under a substitution, then the whole substitution is actually determined by part of it.

LEMMA 9.3. If a typing rule is applicable or pre-satisfied under two substitutions σ_1 and σ_2, then the following hold:

—If the rule is rule (1), then $\sigma_1 = \sigma_2$ if and only if $\sigma_1(\Phi) = \sigma_2(\Phi)$.
—If the rule is any rule but not rule (1) nor rule (9), then $\sigma_1 = \sigma_2$ if and only if $\sigma_1(\Phi) = \sigma_2(\Phi)$ and $\sigma_1(P) = \sigma_2(P)$.
—If the rule is rule (9), then $\sigma_1 = \sigma_2$ if and only if $\sigma_1(\Phi) = \sigma_2(\Phi)$, $\sigma_1(P) = \sigma_2(P)$, and $\sigma_1(P') = \sigma_2(P')$.
—The above properties remain to hold when we replace $\sigma_1 = \sigma_2$ by $\sigma_1 \sqsubseteq \sigma_2$ and $\sigma_1(\Phi) = \sigma_2(\Phi)$ by $\sigma_1(\Phi) \sqsubseteq \sigma_2(\Phi)$.

PROOF. Follows directly from the definitions of individual typing rules. □

By examining each typing rule R, we observe that if the rule is not rule (9), then the satisfaction of the set $\mathcal{A}_R$ is quite independent of the substitution for Φ. For rule (9), the satisfaction of the set $\mathcal{A}_{(9)}$ is dependent of the substitution for Φ, but also related to the satisfaction of the sets $\mathcal{C}_{(7)}$ and $\mathcal{C}_{(8)}$ in rules (7) and (8), respectively. We formally formulate these as the following lemma.

LEMMA 9.4. Assume that a typing rule R is applicable under a substitution σ_1. Let σ_2 be a substitution such that $\sigma_1 \sqsubseteq \sigma_2$ and $\sigma_2(\Phi) \neq \top_M$. Then the following properties hold:

(1) If the rule is not rule (9), then the rule is applicable under σ_2.

(2) If the rule is rule (9), then either
—rule (9) is applicable under σ_2, or
—rule (7) or (8) fails under some substitution σ with $\sigma(\Phi) = \sigma_2(\Phi)$.

(3) If the rule fails under σ_1, then the rule fails under σ_2.

PROOF. (1) Follows easily from the definition of each individual typing rule. Note that the property does not need the condition $\sigma_1(\Phi) \sqsubseteq \sigma_2(\Phi)$.

(2) Since $\sigma_1 \sqsubseteq \sigma_2$ and $\sigma_2(\Phi) \neq \top_M$, we have that $\mathcal{D}om(\sigma_2) = \mathcal{FV}(\mathcal{A}_{(9)}) \cup \{\Phi\}$, $\sigma_2(\Phi) = \phi_2$, $\sigma_2(P) = \sigma_1(P)$, $\sigma_2(X) = \sigma_1(X)$, $\sigma_2(S) = \sigma_1(S)$, and $\sigma_2(P') = \sigma_1(P')$. Since rule (9) is applicable under σ_1, $\sigma_1(\Phi(P)) \neq \bot_P$ and $\sigma_2(\Phi(P)) \neq \bot_P$.
Assume that rule (9) is not applicable under σ_2. Then we need only to consider the following cases:
—In the case where $\sigma_2(\Phi(P)) \neq \sigma_2(VT, ST, SR)$, since $\sigma_1(\Phi(P)) \sqsubseteq \sigma_2(\Phi(P))$, $\sigma_2(\Phi(P)) = \top_P$. In this case, rule (8) fails under every substitution σ with $\sigma(\Phi) = \phi_2$ and $\sigma(P) = \sigma_2(P)$.
—In the case where $\sigma_2(\Phi(P)) = \sigma_2(VT, ST, SR)$ but $\sigma_2(VT(X)) \neq \sigma_2(\langle S \rangle)$, since $\sigma_1(VT(X)) = \sigma_1(\langle S \rangle) \sqsubseteq \sigma_2(VT(X))$, $\sigma_2(VT(X)) = \mathtt{u}$. Thus rule (8) fails under every substitution σ with $\sigma(\Phi) = \sigma_2(\Phi)$ and $\sigma(P) = \sigma_2(P)$.
—If $\sigma_2(\Phi(P')) \neq \sigma_2(VT', ST', SR')$, since $\sigma_1(\Phi(P')) \sqsubseteq \sigma_2(\Phi(P'))$, $\sigma_2(\Phi(P')) = \top_P$. In this case, rule (7) fails under every substitution σ with $\sigma(\Phi) = \sigma_2(\Phi)$ and $\sigma(P') = \sigma_2(P')$.
—Since $\mathit{called}(\sigma_1(S), \sigma_1(SR))$ and $\sigma_1(VT(X)) \sqsubseteq \sigma_2(VT(X))$ hold, we know that $\mathit{called}(\sigma_2(S), \sigma_2(SR))$ must hold.

(3) We only prove the case for rule (8). Other cases are either analogous or trivial. Since $\sigma_1 \sqsubseteq \sigma_2$ and $\sigma_2(\Phi) \neq \top_M$, we have that $\mathcal{D}om(\sigma_2) = \mathcal{FV}(\mathcal{A}_{(8)}) \cup \{\Phi\}$, $\sigma_2(P) = \sigma_1(P)$, $\sigma_2(X) = \sigma_1(X)$, and $\sigma_2(S) = \sigma_1(S)$. Since rule (8) fails under σ_1, $\sigma_1(\Phi(P)) \neq \bot_P$, and if there is a substitution σ'_1 such that $\mathcal{D}om(\sigma'_1) = \mathcal{D}om(\sigma_1) \cup \{VT, ST, SR, S\}$ and $(\sigma'_1)_{|\{\Phi,P,X\}} = \sigma_1$, then either $\sigma'_1(\Phi(P)) \neq \sigma'_1(VT, ST, SR)$ or $\sigma'_1(VT(X)) \neq \sigma'_1(\langle S \rangle)$. Now assume that there is a substitution σ'_2 such that $\mathcal{D}om(\sigma'_2) = \mathcal{D}om(\sigma_2) \cup \{VT, ST, SR, S\}$ and $(\sigma'_2)_{|\{\Phi,P,X\}} = \sigma_2$. We need to prove that either $\sigma'_2(\Phi(P)) \neq \sigma'_2(VT, ST, SR)$ or $\sigma'_2(VT(X)) \neq \sigma'_2(\langle S \rangle)$.
If $\sigma'_1(\Phi(P)) \neq \sigma'_1(VT, ST, SR)$, since $\sigma_1(\Phi(P)) \sqsubseteq \sigma_2(\Phi(P))$, $\sigma'_2(\Phi(P)) = \top_P$. Thus $\sigma'_2(\Phi(P)) \neq \sigma'_2(VT, ST, SR)$. If $\sigma'_1(\Phi(P)) = \sigma'_1(VT, ST, SR)$ but $\sigma'_1(VT(X)) \neq \sigma'_1(\langle S \rangle)$, since $\sigma_1(\Phi(P)) \sqsubseteq \sigma_2(\Phi(P))$, then $\sigma'_2(VT(X)) \neq \sigma'_2(\langle S \rangle)$.

□

Another useful property is that no rules may fail under a substitution containing a legal M-type.

LEMMA 9.5. If ϕ is a legal M-type, then no rules can ever fail under a substitution σ with $\sigma(\Phi) = \phi$.

PROOF. By Definitions 6.2 and 7.1, it is straightforward to check that if a rule fails under a substitution σ with $\sigma(\Phi) = \phi$, then the M-type cannot be a legal M-type. □

9.3 Reachability

We introduce a relation $reach_\phi(p, p')$, which means that address p' is a static successor of address p with respect to M-type ϕ. Intuitively, the relation builds a control graph for dataflow analysis. The relation depends on a given M-type ϕ, enabling the control graph to automatically change with the current M-type computed by dataflow analysis.

Definition 9.6. For an M-type ϕ with $\phi \neq \top_M$ and two addresses p and p', we define that a relation

$$reach_\phi(p, p')$$

holds if and only if there is a typing rule R that is neither rule (1) nor (8), and a substitution σ such that

—rule R is applicable under σ,
—$\sigma(\Phi) = \phi$,
—$\sigma(P) = p$ for the variable P in rule R, and
—$\sigma(A) = p'$ for some $(\Phi(A) \sqsupseteq B) \in \mathcal{S}_R$.

Formally, an M-type is necessary in the definition, since the way rule (9) determines a successor depends on an M-type. Actually, rule (9) determines that $\sigma(P' + 1)$ is a successor of $\sigma(P)$ only when the `jsr` instruction at address $\sigma(P')$ calls subroutine $\sigma(S)$ and the M-type ϕ records that local variable $\sigma(X)$ at address $\sigma(P)$ has type $\sigma(\langle S \rangle)$.

Reachability is preserved under the increase of its defining M-type unless rule (7) or (8) fails under a substitution containing the increased M-type.

LEMMA 9.7. If $reach_\phi(p, p')$ holds for addresses p and p' and an M-type ϕ, then $reach_{\phi'}(p, p')$ holds for every M-type ϕ' with $\phi \sqsubseteq \phi' \neq \top_M$, unless rule (7) or (8) fails under some substitution σ with $\sigma(\Phi) = \phi'$.

PROOF. By Definition 9.6, since $reach_\phi(p, p')$, there are a typing rule R and a substitution σ such that the conditions in Definition 9.6 are satisfied. For typing rules except rules (1), (8), and (9), it is easy to construct a substitution σ' such that $\sigma \sqsubseteq \sigma'$, $\sigma'(\Phi) = \phi'$, and such that all conditions in Definition 9.6 are satisfied. Thus $reach_{\phi'}(p, p')$. For rule (9), following the same arguments as in the proof of Lemma 9.4-2, we have that either $reach_{\phi'}(p, p')$, unless rule (7) or (8) fails under some substitution σ with $\sigma(\Phi) = \phi'$. □

Now we start to formalize the informal explanation in Section 4.2. First of all, we state a lemma that identifies a static execution path of the corresponding subroutine for a given `ret` instruction.

LEMMA 9.8. Assume that iteration $\mathcal{I}$ produces an intermediate M-type ϕ_k. Assume that there is an address p such that

$$\mathtt{mth}(p) = \mathtt{ret}\ x \text{ and } \phi_k(p) = (vt, \_, sr) \text{ with } vt(x) = \langle s \rangle \text{ and } called(s, sr)$$

for some x, vt, sr, and sbr. Then there are addresses $p_1, p_2, \cdots, p_{n+1}$ with $n \geq 0$ such that

—$\mathtt{mth}(p_1) = \mathtt{jsr}\ s$,

—$p = p_{n+1}$,
—for all $i = 1, \cdots, n$, $reach_\phi(p_i, p_{i+1})$, and
—for all $i = 2, \cdots, n+1$, if $\phi_k(p_i) = (\_, \_, sr_i)$ for some sr_i, then $called(s, sr_i)$.

Proof. Since the transfer function corresponding to rule (7) is the only one that can extend a subroutine record and introduce type $\langle s \rangle$ into an intermediate M-type, and since the transfer function corresponding to rule (9) is the only one that can remove some pairs from the end of a subroutine record, by $vt(x) = \langle s \rangle$ and by the definition of iteration $\mathcal{I}$, there are addresses $p_1, p_2, \cdots, p_{n+1}$ with $n \geq 0$ such that $\mathtt{mth}(p_1) = \mathtt{jsr}\ s$, $p_2 = s$, $p_{n+1} = p$, and there are $\phi_{h_1}, \cdots, \phi_{h_n} \in \{\overline{\phi_k}\}$ with $reach_{\phi_{h_i}}(p_i, p_{i+1})$ for all $i = 1, \cdots, n$. Without loss of generality, we may choose the addresses $p_1, p_2, \cdots, p_{n+1}$ such that $\mathtt{mth}(p_i) \neq \mathtt{jsr}\ s$ for all $i = 2, \cdots, n+1$.
Assume that for each $i = 2, \cdots, n+1$, $\phi_{h_i}(p_i) = (\_, \_, sr'_i)$ for some sr'_i.
Furthermore, the typing rules cannot be rule (9) under a substitution σ_i such that $\sigma_i(S) = s$, or $\sigma_i(S) \neq s$ and a pair $(\sigma_i(S), \_)$ occurs in sr'_i before the pair $(s, \_)$. For, in either case, we would have $not(called(s, sr'_{i+1}))$, and since $\mathtt{mth}(p_i) \neq \mathtt{jsr}\ s$ for all $i = 2, \cdots, n+1$, the condition $called(s, sr)$ would be false.
By observing all typing rules that may be possibly used in inducing the relations $reach_{\phi_{h_i}}(p_i, p_{i+1})$, $called(s, sr'_i)$ holds for all $i = 2, \cdots, n+1$. By Lemma 9.1, $\phi_{h_i} \sqsubseteq \phi_k$ for all $i = 2, \cdots, n+1$. Therefore, for all $i = 2, \cdots, n+1$, if $\phi_k(p_i) = (\_, \_, sr_i)$, then $called(s, sr_i)$. $\square$

The above lemma actually implies that checking the condition $called(S, SR)$ in $\mathcal{A}_{(9)}$ is practically redundant in the transfer function corresponding to rule (9).
In Section 4.2, it has been observed that if a local variable is not modified on a static execution path of a subroutine, then under certain conditions a legal M-type records that the type of this local variable only increases or remains unchanged on the path. Now we formally state the observation. The notations $vt_1(x)$ and $vt_{n+1}(x)$ in the following lemma correspond to the notations t'_1 and t'_2 in Figure 7.

Lemma 9.9. Let ϕ be a legal M-type and $p_1, \cdots, p_{n+1}$ addresses such that the following hold for local variables x and x' and subroutine s:

$$\begin{array}{ll} reach_\phi(p_i, p_{i+1}) & \text{for all } i = 1, \cdots, n \\ \mathtt{mth}(p_1) = \mathtt{jsr}\ s & \\ \mathtt{mth}(p_{n+1}) = \mathtt{ret}\ x' & \\ \phi(p_i) = (vt_i, st_i, sr_i) & \text{for all } i = 1, \cdots, n+1 \\ called(s, sr_i) & \text{for all } i = 2, \cdots, n+1 \\ vt_{n+1}(x') = \langle s \rangle & \\ x \notin mvs\_in(s, sr) & \end{array}$$

Then $vt_1(x) \sqsubseteq vt_{n+1}(x)$.

Proof. We use induction on k to prove that $vt_1(x) \sqsubseteq vt_k(x)$ for $1 \leq k \leq n$.
For $k = 1$,[4] the assertion holds trivially. Assume that the assertion holds for all i with $1 \leq i \leq k$. Examining the typing rule that induces $reach_\phi(p_k, p_{k+1})$, we distinguish between two cases.

[4] In reality, $k = 1$ is impossible, since the subroutine needs to have an `astore` instruction to store the returning address to local variable x'. But this point is formally not relevant to the proof.

—If the rule is not rule (9), since ϕ is a legal M-type, by observing each possible typing rule, it is straightforward to see that $vt_k(x) \sqsubseteq vt_{k+1}(x)$. By the induction assumption, $vt_1(x) \sqsubseteq vt_{k+1}(x)$.

—If the rule is rule (9), then $\mathtt{mth}(p_k) = \mathtt{ret}\ x''$ for some local variable x''. Since ϕ is a legal M-type, by rule (8), $vt_k(x'') = \langle s' \rangle$ for some subroutine s'. Since $reach_\phi(p_i, p_{i+1})$ for all $i = 1, \cdots, n$, $\mathtt{mth}(p_1) = \mathtt{jsr}\ s$ and $called(s, sr_i)$ for all $i = 2, \cdots, n+1$, by observing all typing rules, in particular rules (7) and (9), there is h with $1 \leq h < k$ such that $\mathtt{mth}(p_h) = \mathtt{jsr}\ s'$. Note that since ϕ is a legal M-type, rule (7) ensures that $s' \neq s$. Again since ϕ is a legal M-type, by the definition of rule (9), either $vt_k(x) \sqsubseteq vt_{k+1}(x)$ or $vt_h(x) \sqsubseteq vt_{k+1}(x)$. By the induction assumption, $vt_1(x) \sqsubseteq vt_{k+1}(x)$.

□

We are now ready to formalize the key point in Section 4.2, where the notations $vt^{p'}(x)$ and $vt^{p}(x)$ in the following lemma correspond to the notations t_1' and t_2' in Figure 7.

Lemma 9.10. Let p and p' be two addresses, and ϕ_k an intermediate M-type produced by iteration $\mathcal{I}$ and ϕ a legal M-type with $\phi_k \sqsubseteq \phi$ such that

$$\begin{array}{lll} \mathtt{mth}(p) = \mathtt{ret}\ x' & \mathtt{mth}(p') = \mathtt{jsr}\ s & \\ \phi_k(p) = (vt, \_, sr) & vt(x') = \langle s \rangle & called(s, sr) \\ \phi(p) = (vt^p, \_, \_) & \phi(p') = (vt^{p'}, \_, \_) & \end{array}$$

If $x \notin mvs\_in(s, sr)$, then $vt^{p'}(x) \sqsubseteq vt^{p}(x)$.

Proof. By Lemma 9.8, there are addresses $p_1, p_2, \cdots, p_{n+1}$ with $n \geq 0$ such that $\mathtt{mth}(p_1) = \mathtt{jsr}\ s$, $p_2 = s$, $p_{n+1} = p$, $reach_\phi(p_i, p_{i+1})$ for all $i = 1, \cdots, n$, and for all $i = 2, \cdots, n+1$, if $\phi_k(p_i) = (\_, \_, sr_i)$ for some sr_i then $called(s, sr_i)$. Note that it is possible that $p_1 \neq p'$. Since ϕ is a legal M-type, and since $\phi_k \sqsubseteq \phi$ and $reach_\phi(p_i, p_{i+1})$ for all $i = 1, \cdots, n$, by Lemmas 9.7 and 9.5, $reach_\phi(p_i, p_{i+1})$ for all $i = 1, \cdots, n$. By the structure of rule (7), $reach_\phi(p', p_2)$.

Let $p_1' \overset{def}{=} p'$ and $p_i' \overset{def}{=} p_i$ for $i = 2, \cdots, n+1$. Since ϕ is a legal M-type, by Lemma 9.5, $\phi(p_i') = (vt_i', \_, sr_i')$ for all $i = 1, \cdots, n$ and some vt_i' and sr_i'. Note that $vt_1' = vt^{p'}$. The current lemma assumes that $\phi(p) = (vt^p, \_, \_)$. Thus we let $\phi(p_{n+1}') \overset{def}{=} (vt_{n+1}', \_, sr_{n+1}')$ for some sr_{n+1}' and with $vt_{n+1}' \overset{def}{=} vt^p$. Since $\phi_k \sqsubseteq \phi$, $\phi_k(p_{n+1}) = (\_, \_, sr_{n+1})$. Considering the result in the above obtained by applying Lemma 9.8, we have that $called(s, sr_i)$ for all $i = 2, \cdots, n+1$. Since again $\phi_k \sqsubseteq \phi$, $called(s, sr_i')$ for $i = 2, \cdots, n+1$. By Lemma 9.9, $vt^{p'}(x) \sqsubseteq vt^p(x)$. □

9.4 Monotonicity-with-Fixpoint

We prove that procedure $\mathcal{P}$ for each typing rule except rule (9) is monotone.

Lemma 9.11. For all typing rules R except rule (9) and all substitutions σ_1 and σ_2 such that $\mathcal{D}om(\sigma_1) = \mathcal{D}om(\sigma_2) = \mathcal{FV}(\mathcal{A}_R) \cup \{\Phi\}$, $\sigma_1 \sqsubseteq \sigma_2$ and $\sigma_2(\Phi) \neq \top_M$, we have that $\mathcal{P}(\sigma_1, R) \sqsubseteq \mathcal{P}(\sigma_2, R)$.

Proof. If rule R is not applicable under σ_1, then $\mathcal{P}(\sigma_1, R) = \sigma_1(\Phi)$. By Lemma 9.1, $\sigma_2(\Phi) \sqsubseteq \mathcal{P}(\sigma_2, R)$. Since $\sigma_1 \sqsubseteq \sigma_2$, $\mathcal{P}(\sigma_1, R) \sqsubseteq \mathcal{P}(\sigma_2, R)$.

Assume that rule R is applicable under σ_1. Since rule R is not rule (9), by Lemma 9.4-1, rule R is applicable under σ_2.

If there are substitutions σ_i' for $i = 1, 2$ such that rule R is pre-satisfied under σ_i' and $(\sigma_i')_{|\mathcal{FV}(\mathcal{A}_R)\cup\{\Phi\}} = \sigma_i$, then

$$\mathcal{P}(\sigma_i, R) = \sigma_i'(\Phi)[\sigma_i'(A) \mapsto \sigma_i'(\Phi(A)) \sqcup \sigma_i'(B) \mid (\Phi(A) \sqsupseteq B) \in \mathcal{S}_R]$$

for $i = 1, 2$. Examine the typing rules, it is easy to see that $\sigma_1 \sqsubseteq \sigma_2$ implies that $\sigma_1(B) \sqsubseteq \sigma_2(B)$ for each $(\Phi(A) \sqsupseteq B) \in \mathcal{S}_R$. Thus $\mathcal{P}(\sigma_1, R) \sqsubseteq \mathcal{P}(\sigma_2, R)$.

If there is such a substitution σ_1' as above, but no such a σ_2', then $\mathcal{P}(\sigma_1, R)$ is defined as above but $\mathcal{P}(\sigma_2, R) = \top_M$. Thus $\mathcal{P}(\sigma_1, R) \sqsubseteq \mathcal{P}(\sigma_2, R)$.

If there is no such a substitution σ_1' as above, then by Lemma 9.4-3, since $\sigma_1 \sqsubseteq \sigma_2$, there is no such a substitution σ_2' as above, either. In this case $\mathcal{P}(\sigma_1, R) = \mathcal{P}(\sigma_2, R) = \top_M$. □

We prove that procedure $\mathcal{P}$ for rule (9) is monotone-with-fixpoint.

Lemma 9.12. Assume that ϕ_1 is an intermediate M-type produced by iteration $\mathcal{I}$, and assume that ϕ_2 is a legal M-type with $\phi_1 \sqsubseteq \phi_2 \neq \top_M$. Let σ_1 and σ_2 be substitutions such that $\mathcal{D}om(\sigma_1) = \mathcal{D}om(\sigma_2) = \mathcal{FV}(\mathcal{A}_{(9)}) \cup \{\Phi\}$, $\sigma_1 \sqsubseteq \sigma_2$, $\sigma_1(\Phi) = \phi_1$ and $\sigma_2(\Phi) = \phi_2$. Then $\mathcal{P}(\sigma_1, (9)) \sqsubseteq \mathcal{P}(\sigma_2, (9))$.

Proof. If rule (9) is not applicable under σ_1, the proof is similar to that for Lemma 9.11.

Assume that rule (9) is applicable under σ_1. By Lemma 9.4-2, either rule (9) is applicable under σ_2, or rule (7) or (8) fails under a substitution σ with $\sigma(\Phi) = \phi_2$. Since ϕ_2 is a legal M-type, rule (9) must be applicable under σ_2.

Note that $\mathcal{FV}(\mathcal{A}_{(9)}) \subseteq \mathcal{FV}(\mathcal{C}_{(9)})$, and that rule R is pre-satisfied under σ_1 if and only if it is pre-satisfied under σ_2. Since ϕ_2 is a legal M-type, rule R must be pre-satisfied under both σ_1 and σ_2, where

$$\mathcal{P}(\sigma_i, (9)) = \sigma_i(\Phi)[\sigma_i(A) \mapsto (\sigma_i(\Phi(A)) \sqcup \sigma_i(B)) \mid (\Phi(A) \sqsupseteq B) \in \mathcal{S}_{(9)}]$$

for $i = 1, 2$. Since it is known that $\sigma_1(B) \sqsubseteq \sigma_2(B)$ for the $\Phi(A) \sqsupseteq B \in \mathcal{S}_{(9)}$ implies that $\mathcal{P}(\sigma_1, (9)) \sqsubseteq \mathcal{P}(\sigma_2, (9))$, we need only to prove that $\sigma_1(B) \sqsubseteq \sigma_2(B)$ for the $\Phi(A) \sqsupseteq B \in \mathcal{S}_{(9)}$.

Let us use the notations in the definition of rule (9) and the following additional notations for $i = 1, 2$:

$s = \sigma_i(S),\ p = \sigma_i(P),$
$vt_i = \sigma_i(VT),\ st_i = \sigma_i(ST),\ sr_i = \sigma_i(SR),$ (thus $(vt_i, st_i, sr_i) = \sigma_i(\Phi(P))$
$p' = \sigma_i(P'),$
$vt_i' = \sigma_i(VT'),\ st_i' = \sigma_i(ST'),\ sr_i' = \sigma_i(SR'),$ (thus $(vt_i', st_i', sr_i') = \sigma_i(\Phi(P'))$)

Since $\phi_1 \sqsubseteq \phi_2$, we have that

$$vt_1 \sqsubseteq vt_2,\ st_1 \sqsubseteq st_2,\ sr_1 \sqsubseteq sr_2,$$
$$vt_1' \sqsubseteq vt_2',\ st_1' \sqsubseteq st_2',\ sr_1' \sqsubseteq sr_2'.$$

Let us define v_i, k_i, and m_i for $i = 1, 2$ as follows:

$$v_i \stackrel{def}{=} vt'_i[x \mapsto \mathit{filter}(vt_i(x), sbs\_in(s, sr_i)) \mid x \in mvs\_in(s, sr_i)]$$
$$k_i \stackrel{def}{=} \mathit{filter\_l}(st_i, sbs\_in(s, sr_i))$$
$$m_i \stackrel{def}{=} ad\_mvs(mvs\_in(s, sr_i), sr\_bef(s, sr_i))$$

Then we have that $\sigma_i(B) = (v_i, k_i, m_i)$ for the $\Phi(A) \sqsupseteq B \in \mathcal{S}_{(9)}$ and for $i = 1, 2$. Now we need only to prove that $v_1 \sqsubseteq v_2$, $k_1 \sqsubseteq k_2$, and $m_1 \sqsubseteq m_2$.

In order to prove that $v_1 \sqsubseteq v_2$, we prove that $v_1(x) \sqsubseteq v_2(x)$ for each x. We need only to distinguish between three cases for x. Note that since $sr_1 \sqsubseteq sr_2$, $mvs\_in(s, sr_1) \subseteq mvs\_in(s, sr_2)$.

—If $x \notin mvs\_in(s, sr_i)$ for $i = 1, 2$, then $v_i(x) = vt'_i(x)$. By $vt'_1 \sqsubseteq vt'_2$, $v_1(x) \sqsubseteq v_2(x)$.

—If $x \notin mvs\_in(s, sr_1)$ but $x \in mvs\_in(s, sr_2)$, then

$$v_1(x) = vt'_1(x) \text{ and}$$
$$v_2(x) = \mathit{filter}(vt_2(x), sbs\_in(s, sr_2)).$$

Since $x \notin mvs\_in(s, sr_1)$, $\sigma_1(\Phi) \sqsubseteq \sigma_2(\Phi)$, $\sigma_1(\Phi)$ is an intermediate M-type produced by iteration $\mathcal{I}$, rule (9) is applicable under σ_1 (thus $\mathit{called}(s, sr_1)$), and $\sigma_2(\Phi)$ is a legal type, by Lemma 9.10, we have that $vt'_2(x) \sqsubseteq vt_2(x)$. By $vt'_1 \sqsubseteq vt'_2$, $vt'_1(x) \sqsubseteq vt'_2(x)$. Since $vt_2(x) \sqsubseteq v_2(x)$ trivially holds,

$$v_1(x) = vt'_1(x) \sqsubseteq vt'_2(x) \sqsubseteq vt_2(x) \sqsubseteq v_2(x).$$

—If $x \in mvs\_in(s, sr_i)$ for $i = 1, 2$, then

$$v_i(x) = \mathit{filter}(vt_i(x), sbs\_in(s, sr_i)).$$

We distinguish between whether $vt_2(x) = \langle s'' \rangle$ for some s''.

—Assume that it is the case. By $\bot \neq vt_1(x) \sqsubseteq vt_2(x)$, $vt_1(x) = vt_2(x)$. Since $sbs\_in(s, sr_1) = sbs\_in(s, sr_2)$, $v_1(x) = v_2(x)$.

—Assume that it is not the case. If $vt_1(x) = \langle s'' \rangle$ for some s'', then by $vt_1(x) \sqsubseteq vt_2(x)$, $vt_2(x) = \mathtt{u}$. Hence $v_1(x) \sqsubseteq v_2(x)$. If $vt_1(x) \neq \langle s'' \rangle$, then $v_1(x) = vt_1(x) \sqsubseteq vt_2(x) = v_2(x)$.

The proof for $k_1 \sqsubseteq k_2$ is similar to the above case when $x \in mvs\_in(s, sr_i)$ for $i = 1, 2$.

Since $sr_1 \sqsubseteq sr_2$ holds, we have that $mvs\_in(s, sr_1) \subseteq mvs\_in(s, sr_2)$ and that $sr\_bef(s, sr_1) \sqsubseteq sr\_bef(s, sr_2)$. Thus $m_1 \sqsubseteq m_2$ holds. □

We now put the above two lemmas together: if there is a legal M-type, then all intermediate M-types produced by iteration $\mathcal{I}$ are smaller than or equal to this legal M-type.

LEMMA 9.13. Assume that there is a legal M-type ϕ. If iteration $\mathcal{I}$ produces a sequence of intermediate M-types $\phi_0, \phi_1, \cdots$, then $\phi_i \sqsubseteq \phi$ holds for all $i = 0, 1, \cdots$.

PROOF. Initially, $\phi_0 \sqsubseteq \phi$ holds trivially. Assume that $\phi_i \sqsubseteq \phi$ for $0 \leq i \leq k$ and that iteration $\mathcal{I}$ produces ϕ_{k+1}. By the definition of iteration $\mathcal{I}$, $\phi_{k+1} = \mathcal{P}(\sigma_k, R)$ for a typing rule R and a substitution σ_k with $\mathcal{D}om(\sigma) = \mathcal{FV}(\mathcal{A}_R) \cup \{\Phi\}$ and $\sigma_k(\Phi) = \phi_k$. By Lemma 8.1, $\phi = \mathcal{P}(\sigma, R)$ for the same rule and a substitution σ

with $\mathcal{D}om(\sigma) = \mathcal{FV}(\mathcal{A}_R) \cup \{\Phi\}$ and $\sigma(\Phi) = \phi$. If the rule is not rule (9), then by Lemmas 9.11, $\phi_{k+1} = \mathcal{P}(\sigma_k, R) \sqsubseteq \mathcal{P}(\sigma, R) = \phi$. If the rule is rule (9), then by Lemma 9.12, $\phi_{k+1} = \mathcal{P}(\sigma_k, R) \sqsubseteq \mathcal{P}(\sigma, R) = \phi$. □

9.5 Least Legal M-types

THEOREM 9.14 (LEAST LEGAL M-TYPES). *If the method body* `mth` *is well-typed, then iteration $\mathcal{I}$ always computes the least one among of all its legal M-types.*

PROOF. Since the method body `mth` is statically well-typed, it has a legal M-type. By Theorem 9.2, iteration $\mathcal{I}$ is always terminating. By Lemma 9.13, iteration $\mathcal{I}$ must terminate with an M-type ϕ such that $\phi \sqsubseteq \phi'$ for each legal M-type of the method body `mth`. Clearly, $\phi \neq \top_M$. By Theorems 8.2, ϕ is itself an M-type of `mth`. □

COROLLARY 9.15. (1) Iteration $\mathcal{I}$ yields an M-type that is not $\top_M$ if and only if the method body `mth` is well-typed.
(2) Iteration $\mathcal{I}$ yields M-type $\top_M$ if and only if the method body `mth` is not well-typed.

PROOF. Follows directly from Theorems 9.2, 8.2, and 9.14. □

10. RELATED WORK

10.1 Sun's JDK 1.2 Implementation

It is interesting to see how Sun's bytecode verifier in JDK 1.2 treats subroutines. Essentially Sun's bytecode verifier implements something quite similar to the subroutine records defined here. One main difference is that when subroutine records are merged, Sun's bytecode verifier computes the greatest common prefix of the lists of called subroutines in these subroutine records as that for the resulting subroutine record. This means that merging subroutine records may cause some subroutine calls to be removed from some subroutine record, and that after the merging point, any call to the removed subroutine is allowed. The latter point implies that some subroutines may be recursively called. Although one may argue if this special feature is desirable or not, it clearly does not conform to the SJVMS.

In our specification, the definition of the relation $\sqsubseteq$ on subroutine records in Section 5.1 implies that merging two subroutine records with different lists of subroutine calls always leads to a bytecode verification error. Since no subroutines may be removed in the merging process, the condition in rule (7) indeed ensures that no subroutines may be recursively called. From this perspective, our specification is closer to the SJVMS.

Except the above difference, our iteration roughly captures the essence of Sun's bytecode verifier in JDK 1.2 in dealing with subroutines. It is worth pointing out that if needed, our specification can be easily adapted to model the above special feature in Sun's bytecode verifier.

10.2 Formal Specification of Bytecode Verification

Our chaotic iteration is described using a simplified version of the typing rules in Qian [1998]. The work [Qian 1998] proves the soundness of the typing rules with respect to an operational semantics and thus solves a completely different problem from the current paper. It is noteworthy that we could formalize a chaotic iteration without making use of the formulation of existing typing rules, but the proofs might become less direct than they are now.

Stata and Abadi [1998] have proposed a clean typing system for subroutines, provided lengthy proofs for the soundness of the system, and clarified several key semantic issues about subroutines. Freund and Mitchell [1998] have extended Stata and Abadi's system by considering object initialization. Their approach implicitly follows the spirit of dataflow analysis. Thus the typing rules here roughly correspond to some of their rules. At the system level, they use special rules to organize the application of rules for individual instructions, whereas we regard rules directly as constraints. At the instruction level, their typing rules are more abstract than ours: their rules guess the usage of local variables in a subroutine at the `jsr` instruction and use the information as a basis for type inference within the subroutine. Thus additional work is needed to implement the guess step in a verification algorithm. The typing rules here do not guess. Thus one can have an implementation close to the rules, as shown in the current paper.

Similar to the work in this paper, Freund and Mitchell [1999b] (see also [1999a]) have designed a verification algorithm based on Stata and Abadi [1998] and Freund and Mitchell [1998]. To implement the guess step mentioned above, their algorithm pre-computes almost all execution paths and the set of all modified local variables in each subroutine. Following the tradition of Stata and Abadi [1998], the pre-computation computes modified local variables at each `jsr` instruction, instead of at each `ret` instruction. This pre-computation relies heavily on the restriction that no subroutines may be recursively called. After the above phase, the main phase of dataflow analysis can be described as a standard monotone chaotic iteration. In general, their way of breaking the dependence of the main dataflow analysis on the computation of modified local variables is an important contribution toward the full understanding of JVM subroutines.

Potential problems with their approach could be that a multiple-phase algorithm might be farther away from the SJVMS and most commercial bytecode verifiers, and often could lead to a less efficient implementation, than a single-phase one. Since their approach relies heavily on the restriction that no subroutines may be recursively called, it is unclear how easily their approach can be adapted to model the special feature in Sun's bytecode verifier mentioned above.

In addition, we are a little concerned about some of the assumptions made in Freund and Mitchell [1999b], in particular, the assumption that no dead code is present in the JVM programs. The problem is that a `ret` instruction may introduce dead code if it returns to an outer subroutine call. See the code in Figure 13, where the inner subroutine s' directly returns to the caller of subroutine s, and thus causes the `astore` instruction at address $s + 2$ to become dead code. Obviously dead code should not contribute to the set of modified local variables, but the algorithm in Freund and Mitchell [1999b] does not directly address this case. Extending the algorithm to consider this seems possible. The question is how such an extension would complicate matters.

Hagiya and Tozawa [1998] have presented a very original typing system for subroutines and provided an extremely simple soundness proof. They introduce special types indicating which local variables hold values from outer subroutines, instead of

```
p: jsr s          // call a subroutine
   ...
s: astore x       // store return address
   jsr s'         // call nested subroutine
   astore x'      // modify local variable
   ...
s': ret x         // return to outer subroutine
```

Fig. 13. Dead code introduced by a `ret` instruction.

recording which variables are modified. They also introduce special types indicating the position of the subroutine in a subroutine call stack to which a verb+ret+ instruction returns. In addition, their approach also heavily relies on the restriction that no subroutines may be called recursively. Since their idea is quite different from the standard approach, more studies are needed to understand all possible consequences for the full JVM. In particular, it is unclear if their approach can be easily adapted to model the special feature in Sun's bytecode verifier mentioned above. Hagiya and Tozawa have informally outlined a verification algorithm corresponding to their specification but not given any formal description or formal proofs. Therefore, it is difficult to compare their algorithm with ours on a formal basis.

Goldberg [1998] has directly used dataflow analysis to specify bytecode verification and successfully formalized a way to integrate some aspects of class loading into bytecode verification. His specifications can be regarded as a verification algorithm. However, since he did not consider subroutines, he did not encounter the problem we have considered here. His and our approach are complementary.

Pusch [1999] has successfully formalized bytecode verification, and mechanically proved the soundness using the theorem prover Isabelle/HOL. Her formalization is based on Qian [1998]. Mechanical proofs have the advantages that the proofs are more reliable and can be mechanically repeated when small changes are made on some parts of the whole specification. Her work shows that formal tools can be used to model real-life programming languages.

Recently, Nipkow [2000] has formalized and proved correct an executable bytecode verifier using the theorem prover Isabelle/HOL. His formalization follows the style of Kildall's algorithm. He approach is first to define an abstract framework for proving correctness of a class of data flow algorithms, and then instantiate the framework with a model of the JVM. Although some features of the JVM-like subroutines are still missing, his work seems to form a promising step toward a complete formal machine-checked JVM.

O'Callahan [1999b] has constructed a typing system based on polymorphic recursion and continuations similar to a more general setting of typed assembly language as in Morrisett et al. [1998;1998], and compared it with bytecode verification. He reveals that return addresses can be directly typed using continuations so that one does not have to analyze which subroutine each instruction belongs to. He shows that disallowing recursion in return address types can prevent a return address to be used more than once. Unfortunately, bytecode verification corresponds to a problem of generating type information in his typing system, where the decidability of the problem is in general doubtful. Thus some restrictions might be needed to make it decidable. One such restriction would be to disallow return address types from being nested beyond a certain depth [O'Callahan 1999a].

Jones [1998] and Yelland [1999] have independently specified the semantics of JVM instructions using the functional language Haskell. Their specifications are executable programs. Thus part of them can be regarded as a verification algorithm. Since in their approach composition of JVM instructions corresponds to composition of functional programs, their specifications are particularly flexible when changing or extending the JVM instruction set. The soundness of the specification, as explained in Yelland [1999], can be established indirectly using known semantic foundations for Haskell, although it might be difficult to understand for people unfamiliar with these foundations. In general, both the techniques and the allowed programs in their work are quite different from those described in the SJVMS and in most actual implementations, while the techniques and the allowed programs in our work are much closer. They have not identified any nonmonotonicity property nor explicitly considered operational properties of their verification algorithms as done here. More investigations are needed to see whether their verification algorithms have properties similar to those we have proved here.

Bertelsen [1997] has formalized the JVM using state transitions. Cohen [1997] has described a formal semantics of a subset of the JVM, where runtime checks are used to assure type-safe execution. Both papers have not considered bytecode verification. Recently, Börger and Schulte [1999] have presented a quite comprehensive high-level definition of JVM in a similar style as Bertelsen [1997] and Cohen [1997] and derived a bytecode verifier from the high-level definition. But they have not considered any properties of their bytecode verifier like those discussed in the current paper.

10.3 Fixpoint Theorems and Chaotic Fixpoint Iteration

Fixpoint theorems have taken many forms and been (re-)proved many times in the literature (see Lassez et al. [1982] for a survey). There has been much work on applications of fixpoint theorems in dataflow analysis (e.g., see Kildall [1973] and Muchnick [1997]) and abstraction interpretation (e.g., see Cousot and Cousot [1977]). From these perspectives, our chaotic iteration is not something that is substantially new. However, the result of the current paper is nontrivial, since it relies on the choice of a special form of fixpoint theorem and the application of the theorem using special properties of the JVM instructions.

We are not aware of any paper in the literature directly applying Theorem 2.2. The work by Geser et al. [1996] might be the closest one. It presents a fixpoint theorem that is the same as Theorem 2.2 except that the requirement on monotonicity-with-fixpoint is replaced by that on delay-monotonicity. For a poset $\langle D, \sqsubseteq \rangle$, a set $\mathcal{F}$ of functions from D to D is called *delay-monotone* if for each $f \in \mathcal{F}$ and all elements $d, d' \in D$, $d \sqsubseteq d'$ implies that there is a sequence of (possibly identical) functions $f_1, \cdots, f_m$ with $f_i \in \mathcal{F}$ for all $i = 1, \cdots, m$ such that $f(d) \sqsubseteq f_1(\cdots f_m(d'))$.

In fact, the conditions in Theorem 2.2 as a whole imply delay-monotonicity, and the conditions in the theorem by Geser et al. imply monotonicity-with-fixpoint. To prove the former statement, consider $d \sqsubseteq d'$ and $f \in \mathcal{F}$. By increasingness, the chaotic iteration yields a fixpoint of $\mathcal{F}$ starting from every element. Thus there is a sequence $f_1, \cdots, f_m$ with $f_i \in \mathcal{F}$ for all $1 \leq j \leq m$ such that $f_1(\cdots f_m(d'))$ is a fixpoint. Again by increasingness, $d \sqsubseteq d' \sqsubseteq f_1(\cdots f_m(d'))$. By monotonicity-with-fixpoint, $f(d) \sqsubseteq f_1(\cdots f_m(d'))$. Thus $\mathcal{F}$ is delay-monotone. To prove the latter statement, consider $d \sqsubseteq d'$, where d' is a fixpoint of of $\mathcal{F}$, and $f \in \mathcal{F}$. Since $\mathcal{F}$ is delay-monotone, $f(d) \sqsubseteq f_1(\cdots f_m(d'))$ for a sequence $f_1, \cdots, f_m$. Since d' is a fixpoint, $f_1(\cdots f_m(d')) = d'$ and thus $f(d) \sqsubseteq d'$. Hence $\mathcal{F}$ is monotone-with-fixpoint.

11. CONCLUSION

The control structure of our chaotic iteration is simple, natural, and standard. Since our chaotic iteration does not rely on a special strategy to visit program points, it formally models a core of most existing bytecode verifiers which are based on standard fixpoint computation.

We have proved that the chaotic iteration does compute the sharpest type information for input JVM programs. That is, if an input program is well-typed, then the iteration computes the least M-type; if not, then the iteration yields the artificial top element $\top_M$ in the semilattice.

There are a number of commercial bytecode verifiers in use. It is expected that more bytecode verifiers will be produced in the near future in connection with building different versions of the JVM for various application areas such as database systems, embedded systems, etc. All existing commercial verifiers we know of are based on standard fixed-point computation. The fact that the chaotic iteration computes the sharpest type information suggests that it is theoretically possible to let all such bytecode verifiers accept the same set of well-typed JVM programs. In other words, if a JVM program has been determined to be well-typed using a verifier on a server host, then it should be determined to be well-typed by every (possibly different) verifier on a client host. The result clearly increases the confidence in the availability of services provided by mobile code in a network equipped with different bytecode verifiers.

Currently, we are using the Specware system to formally specify bytecode verification and synthesize an algorithm from that specification [Coglio et al. 1998]. The synthesized verification algorithm directly corresponds to that in the current paper except that for practical reasons we are synthesizing concrete transfer functions for individual instructions, instead of a generic one like procedure $\mathcal{P}$.

ACKNOWLEDGMENTS

We sincerely thank Alessandro Coglio and Allen Goldberg for intensive discussions on related issues, and Lindsay Errington for comments on the paper. We also thank Sophia Drossopoulou for comments on an early version of the paper, and Phillip Yelland and Robert O'Callahan for explaining their related work.

REFERENCES

Bertelsen, P. 1997. Semantics of Java byte code. Available under `ftp://ftp.dina.kvl.dk/pub/Staff/Peter.Bertelsen/jvm-semantics.ps.gz`.

Börger, E. and Schulte, W. 1999. Modular design for the Java virtual machine architecture. `ftp://ftp.di.unipi.it/pub/Papers/boerger/jvmarch.ps`.

Coglio, A., Goldberg, A., and Qian, Z. 1998. Towards a provably-correct implementation of the JVM bytecode verifier. In *Proc. OOPSLA'98 Workshop Formal Underpinnings of Java*. IEEE Computer Society Press.

COHEN, R. 1997. The Defensive Java Virtual Machine specification. Tech. rep., Computational Logic inc.

COUSOT, P. AND COUSOT, R. 1977. Abstract interpretation: a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proc. 4th ACM Symp. Principles of Programming Languages.* ACM, New York, 238–258.

COUSOT, P. AND COUSOT, R. 1979. Constructive versions of Tarski's fixed point theorems. *Pacific Journal of Mathematics 82,* 1, 43–57.

FREUND, S. AND MITCHELL, J. 1998. A type system for object initialization in the Java bytecode language. In *Proc. ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications.* ACM, New York, 310–328.

FREUND, S. AND MITCHELL, J. 1999a. A formal framework for the Java bytecode language and verifier. In *Proc. ACM Conf. on Object-Oriented Programming, Systems, Languages, and Applications.* ACM, New York, 147–166.

FREUND, S. AND MITCHELL, J. 1999b. Specification and verification of Java bytecode subroutines and exceptions. Tech. rep., Computer Science Department, Stanford University.

GESER, A., KNOOP, J., LÜTTGEN, G., RÜTHING, O., AND STEFFEN, B. 1996. Non-monotone fixtpoint iterations to resolve second order effects. In *Proc. 6th Int. Conf. on Compiler Construction.* Springer-Verlag LNCS 1060, 106–120.

GOLDBERG, A. 1998. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conference on Computer and Communications Security.* ACM, New York.

HAGIYA, M. AND TOZAWA, A. 1998. On a new method for dataflow analysis of Java Virtual Machine subroutines. In *Proc. 1998 Static Analysis Symposium.* Springer-Verlag LNCS.

JONES, M. 1998. The functions of Java bytecode. In *Proc. OOPSLA'98 Workshop Formal Underpinnings of Java.*

KILDALL, G. 1973. A unified approach to global program optimization. In *Proc. ACM Symp. Principles of Programming Languages.* ACM, New York.

LASSEZ, J.-L., NGUYEN, V., AND SONNENBERG, E. 1982. Fixed point theorems and semantics: A folk tale. *Information Processing Letters 14,* 3, 112–116.

LINDHOLM, T. AND YELLIN, F. 1996. *The Java Virtual Machine Specification.* Addison-Wesley.

MORRISETT, G., CRARY, K., GLEW, N., AND WALKER, D. 1998. Stack-based typed assembly language. In *ACM Workshop for Types in Compilation.*

MORRISETT, G., WALKER, D., CRARY, K., AND GLEW, N. 1998. From system F to typed assembly language. In *Proc. 25nd ACM Symp. Principles of Programming Languages.* ACM, New York, 85–97.

MUCHNICK, S. 1997. *Advanced Compiler Design and Implementation.* Morgan Kaufmann, San Francisco.

NIPKOW, T. 2000. Verified bytecode verifiers. Tech. rep., Technische Universität München. `http://www.in.tum.de/ nipkow/pubs/bcv2.html`.

O'CALLAHAN, R. 1999a. Private communication.

O'CALLAHAN, R. 1999b. A simple, comprehensive type system for Java bytecode subroutines. In *Proc. 26th ACM Symp. Principles of Programming Languages.* ACM, New York, 70–78.

PUSCH, C. 1999. Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL. In *Proc. 1999 Conf. on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99).* LNCS, vol. 1579. Springer-Verlag, 89–103.

QIAN, Z. 1998. A formal specification of Java virtual machine instructions for objects, methods and subroutines. In *Formal Syntax and Semantics of Java*, J. Alves-Foss, Ed. LNCS, vol. 1523. Springer-Verlag.

SMOLKA, G., NUTT, W., GOGUEN, J., AND MESEGUER, J. 1989. Order-sorted equational computation. In *Resolution of Equations in Algebraic Structures, volume 2*, H. Aït-Kaci and M. Nivat, Eds. Academic Press, 297–367.

STATA, R. AND ABADI, M. 1998. A type system for Java bytecode subroutines. In *Proc. 25th ACM Symp. Principles of Programming Languages.* ACM, New York.

YELLAND, P. 1999. A compositional account of the Java Virtual Machine. In *Proc. 26th ACM Symp. Principles of Programming Languages.* ACM, New York, 70–78.

Received June 1999; revised December 1999; accepted June 2000