Constraint-Based Specification and Dataflow Analysis for Java™ Bytecode Verification

Zhenyu Qian

Kestrel Institute, 3260 Hillview Avenue, Palo Alto, CA 94304, USA*
Aug. 10, 1998

Abstract. (Java) bytecode verification should prevent various runtime errors in Java Virtual Machine (JVM) programs and play an important part in ensuring Java-based internet security. The official JVM specification is inadequate for security-critical applications. Recent research work has proposed using formal typing systems to specify bytecode verification. However, it is still unknown how to make an implementation, whose correctness and completeness with respect to the formal specification can be proved.

This paper proposes an approach to bridging the gap. We first give formal typing rules enforcing formal constraints on types for memory locations in simplified legal JVM programs. Then we present a dataflow analysis algorithm (scheme) as a bytecode verifier, which non-deterministically uses the formal rules to compute the smallest types for memory locations that satisfy the constraints, or to report a failure if the computation fails. By formally proving the correctness, termination and completeness, we show a way to reach a bytecode verifier, which is provably correct with respect to a formal specification.

1 Introduction

Java Virtual Machine (JVM) is designed so that JVM programs (i.e. *bytecode*) can be dynamically loaded over the Web and run locally. In order to ensure that no security problems may arise in the local system, mechanisms are needed to enforce restrictions on dynamically loaded bytecode. One of the mechanisms is *bytecode verification*, which should statically ensure that bytecode will never use data with wrong types in execution. Although the Java industry has been working hard to make their bytecode verifiers secure, several serious flaws have been discovered (and fixed) so far (see e.g. [10,15]).

The problem has at least two aspects. The first is to ask whether the specification is adequate to guarantee the security. The official JVM specification (OJVMS) is inadequate, since it is ambiguous and hard to reason about. Therefore, recent research work (e.g. [17,5,13]) proposes using formal typing systems to specify bytecode verification.

* The author gratefully acknowledges the support by DARPA contract F30602-96-C-0363. Part of the work was done while the author was at Bremen Institute for Safe Systems, Department of Computer Science, University of Bremen, Germany.

The second aspect of the problem is that even if there is a formal specification, it is still unknown how to make an implementation, whose correctness and completeness with respect to the formal specification can be proved.

This paper proposes an approach to solving the second aspect of the problem. We give formal typing rules enforcing constraints on types for memory locations in simplified legal JVM programs (in Section 5). Then we present a dataflow analysis algorithm (scheme) as a bytecode verifier (in Section 7), which non-deterministically uses the formal typing rules to compute the smallest types for memory locations that satisfy the constraints, or to report a failure if the computation fails. By formally proving the correctness, termination and completenessof the algorithm in Section 8, we show a way to reach a bytecode verifier, which is provably correct with respect to a formal specification.

2 Overview of our approach

JVM instructions are assembly instructions mainly operating on a stack, called the *operand stack*, and occasionally on registers, called *local variables*. Operand stack entries and local variables are also called *local memory locations*. Data in local memory locations are either object references or primitive values. The full JVM uses a heap to store objects. But we do not consider it in this paper for simplicity.

A *JVM method* contains, among others, the name of the method, the types of all parameters and return value, the number of local variables, and a sequence of JVM instructions, called the *body*. The position of a JVM instruction in a body is called a *program point*. A *JVM class* (i.e. *class file*) is the result obtained by compiling a Java class. A class file contains a JVM method if and only if the original Java class contains the corresponding Java method.

In this paper we consider the following instructions:

- An instruction `aload` *ind* loads an object, and an instruction `iload` *ind* an integer value, from the local variable *ind* onto the operand stack.
- An instruction `astore` *ind* stores an object, and an instruction `istore` *ind* an integer value, from the operand stack into the local variable *ind*.
- An instruction `goto` *pp* transfers control to the program point *pp*.
- An instruction `if_acmpeq` *pp* compares the two top object references in the operand stack, and if equal, then transfers control to the program point *pp*, else to the next program point.
- We assume that an implicit instruction at the start of a method sets an empty stack as the operand stack, and stores the object, on which the method is invoked, and all actual parameters into the first local variables.
- An instruction `return` terminates the current method.
- Instructions `jsr` and `ret` build the mechanism for subroutines. More concretely, `jsr` *pp calls* the *subroutine pp* by first pushing the next program point as the return program point onto the operand stack and then

transferring control to the program point pp, while `ret` ind returns from a subroutine by using the content of the local variable ind as a return program point and transferring control to it. A subroutine need not have a `ret`. If it has, then an `astore` should be used to store the return program point that has been pushed onto the operand stack by `jsr` into a local variable so that `ret` can access it.

2.1 An example: the state transition

Consider, for example, the method in Figure 1, which has the name `m`, two parameters of the primitive type `int`, no return value, and 5 local variables.

To illustrate the operational semantics, Figure 1 also gives all states of local memory locations in an execution. The states of local variables are recorded as lists of the same length under "L", those of the operand stack as lists of different lengths under "S".

Program	L	S	Successor
`Method void m(int,int)`			
`.limit locals 5`			
`0 jsr 5`	$[obj, val1, val2, \_, \_]$	$[]$	5
`1 aload 0`	$[obj, val1, val2,1,12]$	$[]$	2
`2 astore 2`	$[obj, val1, val2,1,12]$	$[obj]$	3
`3 jsr 5`	$[obj, val1, obj,1,12]$	$[]$	5
`4 return`	$[obj, val1, obj,4,12]$	$[]$	returns
`5 astore 3`	$[obj, val1, val2\langle obj\rangle, \_, \_]$	$[1\langle 4\rangle]$	6
`6 aload 0`	$[obj, val1, val2\langle obj\rangle, 1\langle 4\rangle, \_]$	$[]$	7
`7 aload 0`	$[obj, val1, val2\langle obj\rangle, 1\langle 4\rangle, \_]$	$[obj]$	8
`8 if_acmpeq 11`	$[obj, val1, val2\langle obj\rangle, 1\langle 4\rangle, \_]$	$[obj, obj]$	11
`9 aload 0`			
`10 astore 1`			
`11 jsr 12`	$[obj, val1, val2\langle obj\rangle, 1\langle 4\rangle, \_]$	$[]$	12
`12 astore 4`	$[obj, val1, val2\langle obj\rangle, 1\langle 4\rangle, \_]$	$[12]$	13
`13 ret 3`	$[obj, val1, val2\langle obj\rangle, 1\langle 4\rangle,12]$	$[]$	$1\langle 4\rangle$

Figure 1: A JVM program and an execution

The notation obj stands for the reference of the object on which the method is invoked. The notations $val1$ and $val2$ stand for the two integer actual parameters. The notation "$\_$" in a local memory location means that the state is not important.

We explain some details. First, due to the implicit instruction, the local variables at the program point `0` have the state $[obj, val1, val2, \_, \_]$, meaning that the local variables 0, 1 and 2 hold obj, $val1$ and $val2$, respectively.

The `jsr 5` at `0` calls the subroutine `5`. Note that the subroutine is called both at `0` and `3`. The notation $[obj, val1, val2\langle obj\rangle, \_, \_]$ at `5` means that the local variables have the state $[obj, val1, val2, \_, \_]$ when the subroutine is entered from `0`, and $[obj, val1, obj, \_, \_]$ when entered from `3`. In Figure 1, whenever a local memory location holds different data in different executions of the subroutine `5`, we use $\langle\rangle$ to quote the one in the execution entered from `3`.

The `astore 3` at `5` yields a state at `6`, which is the state at `5` except that the local variable 3 gets the top entry of the operand stack.

The `ret 3` at `13` uses the content in the local variable 3 as the return program point. In the two executions of the subroutine `5`, the return program points are `1` and `4`, respectively. The states yielded at `1` and `4` are equal to the state of the corresponding execution at `13`. Note that the subroutine `5` calls another subroutine `12`, but the latter directly returns to the caller of the former via the `ret 3` at `13`.

2.2 An example: static types of local memory locations

Bytecode verification should assign a kind of type, called a *static type*, to each local memory location at every program point. Roughly speaking, a static type is either directly a type of some runtime datum or the union of several types of runtime data. We call a mapping from pairs of local memory locations and program points to static types a *program type*.

A JVM program is *legal* if and only if there is a program type for it such that all constraints on the static types of local memory locations enforced by our typing rules are satisfied.

The method in Figure 1 is a legal JVM program, where static types assigned to local variables and operand stack entries are given as lists in the columns LT and ST, respectively, in Figure 2.

We assume that the method is contained in a class `C`. Then the static types involved are C, *int*, *unus* and $sbr(pp)$, where *unus* is the static type of all data and $sbr(pp)$ is the static type of the return program points for all occurrences of the instruction `jsr` *pp*calling the subroutine *pp*. Note that $sbr(pp) \neq sbr(pp')$ if and only if $pp \neq pp'$. Since *unus* is the static type of all data, including data of *int*, C and $sbr(pp)$, local memory locations with the static type *unus* are unusable (by most instructions)[1].

These static types together with an artificial element $\perp$ build a lattice, whose partial ordering $\sqsupseteq$ satisfies that $unus \sqsupseteq C \sqsupseteq \perp$, $unus \sqsupseteq int \sqsupseteq \perp$ and $unus \sqsupseteq sbr(pp) \sqsupseteq \perp$. For any static types *stty* and $stty'$, we say that *stty covers* $stty'$, if $stty \sqsupseteq stty'$.

The column "Successors" contains all statically possible successor program points of each program point. They will help the discussion, since most typing rules enforce, among others, constraints on the static types at statically possible successor program points.

[1] In the full JVM, some stack manipulation instructions like pop can handle a local memory location with the static type *unus*.

Program	LT	ST	Successors
`Method void m(int,int)`			
`.limit locals 5`			
`0 jsr 5`	$[C,int,int,unus,unus]$	$[]$	5
`1 aload 0`	$[C,unus,int,unus,unus]$	$[]$	2
`2 astore 2`	$[C,unus,int,unus,unus]$	$[C]$	3
`3 jsr 5`	$[C,unus,C,unus,unus]$	$[]$	5
`4 return`	$[C,unus,C,unus,unus]$	$[]$	returns
`5 astore 3`	$[C,unus,unus,unus,unus]$	$[sbr(5)]$	6
`6 aload 0`	$[C,unus,unus,sbr(5),unus]$	$[]$	7
`7 aload 0`	$[C,unus,unus,sbr(5),unus]$	$[C]$	8
`8 if_acmpeq 11`	$[C,unus,unus,sbr(5),unus]$	$[C,C]$	11,9
`9 aload 0`	$[C,unus,unus,sbr(5),unus]$	$[]$	10
`10 astore 1`	$[C,unus,unus,sbr(5),unus]$	$[C]$	11
`11 jsr 12`	$[C,unus,unus,sbr(5),unus]$	$[]$	12
`12 astore 4`	$[C,unus,unus,sbr(5),unus]$	$[sbr(12)]$	13
`13 ret 3`	$[C,unus,unus,sbr(5),sbr(12)]$	$[]$	1,4

Figure 2: Static types for local memory locations of the method in Figure 1

The typing rule for the implicit instruction at the beginning of the method enforces the constraint that static types of all local variables at the program point `0` should cover the static types $[C,int,int,unus,unus]$. We actually assign these static types to the local variables. The static types *unus* for the local variables 3 and 4 ensure that the local variables are unusable at `0`.

The typing rule for `jsr` at `0` and `3` enforces the constraint that static types for local memory locations at `5` should cover the corresponding ones at `0` and at `3`, except that the operand stack should be extended by a static type covering $sbr(5)$. As an example, we assign the static type *unus* to the local variable 2 at `5`, which covers the corresponding static types *int* at `0` and C at `3`.

The typing rule for `astore 3` at `5` enforces the constraint that a static type of the top entry of the operand stack at `5` should be a class[2], and that a static type of each local memory location at `6` should cover that at `5`, except that a static type of the local variable 3 at `6` should cover that of the top entry in the operand stack at `5`.

Before we continue, let us consider a problem in typing subroutines. The problem is because one may wish to let some data at a call site of a subroutine be used after the subroutine returns. Thus if a local variable, say the local variable 2, has different static types *int* and C at different call sites `0` and `3`, respectively, then the local variable 2 in the subroutine should have a static type, *unus* in this case, covering the static type at each calling site. Now if we followed a naive dataflow approach by letting the local variable 2 keep the

[2] In general, it can be a set of classes. See the formal treatment later.

static type *unus* after the subroutine returns, then the local variable 2 would be unusable after the subroutine returns. In other words, if we inserted e.g. `iload 2` at `1`, which should load an integer value from the local variable 2, or `aload` at `4`, which should load an object from the local variable 2, then the JVM program would become illegal.

To overcome this difficulty, the OJVMS requires that if a local variable may not be modified in a subroutine, then the static type of the local variable at each return program point should cover that at the corresponding call site, independent of that before the subroutine returns.

Bearing this requirement in mind, one can better understand the constraints enforced by the typing rule for the `ret 3` at `13`:

1. The local variable 3 at `13` should have a static type of the form $sbr(pp)$, to ensure that it holds a return program point. In this example, $pp =$ `5`. Since `jsr 5` occurs at `0` and `3`, both `1` and `4` are possible return program points. Thus both `1` and `4` are possible successor program points.
2. For each local memory location, we have two cases:
   (a) If it is a local variable that may not be modified in the subroutine `5`, e.g. it is the local variable 2, then its static type at each successor program point should cover that at the corresponding call site, i.e. *int* at `0` or C at `3`. We assign *int* and C to the local variable 2 at `1` and `4`, respectively.
   (b) Otherwise, its static type at each successor program point should cover that at the `ret 3`, i.e. at `13`, except that if the static type at `13` is of the form $sbr(pp')$ for a subroutine pp' called between the corresponding `jsr 5` and the `ret 3`, then the static type at the successor program point should be *unus*. The last except-case enforces that the local variables 3 and 4 at `1` and `4` have *unus*. It serves to assure an OJVMS's requirement, saying that no return program points for subroutines called between a `jsr 5` and the `ret 3` are usable after the subroutine `5` returns. In fact, this detail does not essentially affect the main issue of this paper. But we still include it here for the completeness of the example.

In general, a typing rule statically enforces the constraint that if an instruction uses a local memory location at a program point, then the instruction must be able to deal with all data of the assigned static type. Thus in order to statically ensure that no data with wrong types will be used in the execution of a legal JVM program, we need only to prove the followingtype safety property:

– A local memory location at a program point may only hold data of the assigned static type.

Comparing the states in Figure 1 and the static types in Figure 2, we can intuitively see that this property holds. Indeed, the paper [13] formally shows it for a set of JVM instructions including those given here. Therefore we can take the typing rules given here as a sound specification and use them for further development.

2.3 An example: dataflow analysis

First of all, a lattice of program types can be built from the lattice of static types in a more or less standard way, where the partial order and the meet and join operations in the lattice of program types are obtained componentwise from those in the lattice of static types. Then it does not seem difficult to write a dataflow analysis algorithm, which for every JVM program, either computes a program type such that all constraints enforced by typing rules are satisfied, or reports a failure. The algorithm can be a fixed-point iteration, which starts with the bottom element in the lattice of program types, repeatedly calls some monotone transfer functions yielding an ascending chain of program types, until no new program types may be yielded, or a failure is reported.

The challenge here is to write an algorithm so that its correctness and completeness properties with respect to the typing rules can be formally proved.

Our idea is to have each transfer function being based on each typing rule and to have a non-deterministic fixed-point iteration scheme repeatedly and non-deterministically calling the transfer functions. Intuitively, a transfer function should take a program type as argument, and then determine a program point, check the enforced constraint on the static types at that program point, and yield a program type bigger than the argument such that the enforced constraints on the static types at each successor program point are satisied.

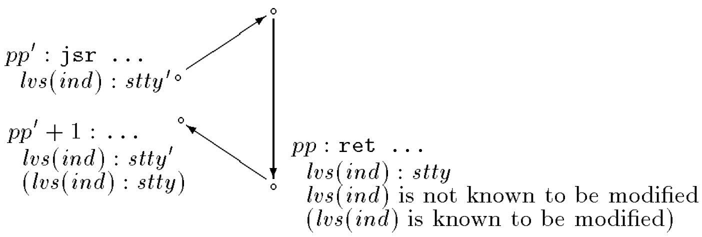

Figure 3: Transfer function for `ret`

The idea works well for all typing rules except for a typing rule for `ret`. The reason is that a naive transfer function built based on the typing rule for `ret` is not monotone in general. The situation is illustrated in Figure 3, where we consider the program point pp of a `ret`, the program point pp' of a corresponding calling `jsr` and the return program point $pp'+1$, and use $lvs(ind) : stty$ and $lvs(ind) : stty'$ to denote that the local variable ind has a static type $stty$ and $stty'$, respectively. Note that the typing rule needs to know which local variables may be modified in the subroutine. A natural way to compute this information is to let each transfer function accumulate the indices of those local variables that are *known* to be modified from a calling `jsr` to the current program point step by step. In Figure 3, assume that the local variable *ind* was first unknown to be modified from pp' to pp. Then its static type at $pp'+1$ can be (in general, should cover) its static type $stty'$ at pp'. However, as soon as the local variable *ind* becomes known to be modified from pp' to pp, its static type at $pp'+1$ should be changed so that it covers the static type $stty$ at pp. In this case, if $stty \sqsupseteq stty'$ does not hold, then the transfer function yields a program type that does not cover the argument program type at $pp'+1$, and thus is not monotone.

One may wonder why we don't first finish computing all modified local variables before starting the main fixed-point iteration. The reason is that to compute all modified local variables, we need to know all possible execution paths. But an execution path may depend on successor program points of a `ret`, which in turn depend on the static type of the local variable that the `ret` uses. Thus it is in general impossible to compute all modified local variables before part of a program type is known.

To ensure the monotonicity property, we propose adding an additional applicability condition requiring that the transfer function for `ret` can be called only when for each local variable that is not known to be modified, its static type at the calling `jsr` covers its static type at the `ret`. This implies that, in Figure 3, if the local variable *ind* is unknown to be modified from pp' to pp, then $stty \sqsupseteq stty'$ is the additional applicability condition for the local variable *ind* at $pp'+1$ to obtain the static type $stty'$.

What remains to be done is to prove that the additional applicability condition does not affect the completeness property. For doing this in Figure 3, we prove that if $lvs(ind)$ is unknown to be modified from pp' to pp, and if $stty \sqsupseteq stty'$ does not hold, then some transfer functions must be able to be applied at a program point other than pp on a path from $+pp'$ to pp to yield a new program type. Indeed, if $lvs(ind)$ is unknown to be modified from pp' to pp, then a path from pp' to pp exists, on which $lvs(ind)$ is not modified. Now if no transfer functions applied at a program point other than pp on the path can yield a new program type, then all enforced constraints on the static types on the path are satisfied. This implies that for each typing rule on the path, the static type of $lvs(ind)$ at a successor program point always covers that at the current program point. By the transitivity property, $stty \sqsupseteq stty'$ must hold. A contradiction arises!

Note that although the intuition appears easy, the formal proof is not easy at all. One of the difficulties is, as mentioned, that successor program points of a `ret` depend on the static type of a local variable.

3 Preliminaries

We use the notation $\overline{\alpha_n}$ to denote a sequence of n syntactical objects $\alpha_1, \cdots, \alpha_n$, and the notation $\{\cdots\}$ a set.

We use $\{\overline{\alpha_n \mapsto \alpha'_n}\}$, where $\alpha_i \neq \alpha_j$ hold for all i, j with $0 \leq i \neq j \leq n$, to denote a mapping, where the mapping of each α_i is α'_i, and the mapping of other elements will yield a *failure* if it is not otherwise stated. We define $\mathcal{D}om(\{\overline{\alpha_n \mapsto \alpha'_n}\}) = \{\overline{\alpha_n}\}$. For a mapping θ and a set E, we use $\theta_{|E}$ to denote $\{\alpha \mapsto \theta(\alpha) \mid \alpha \in \mathcal{D}om(\theta) \cap E\}$. For a mapping θ, we use $\theta(\alpha)$ to denote the result of the mapping for α, and write $\theta[\alpha \mapsto \alpha']$ for the mapping that is equal to θ except it maps α to α'.

A *list* $[\alpha_0 \cdots, \alpha_n]$ is a special mapping $\{i \mapsto \alpha_i \mid 0 \leq i \leq n\}$. We define $[\alpha_0 \cdots, \alpha_n] + \alpha := [\alpha_0 \cdots, \alpha_n, \alpha]$.

A *lattice* is a 4-tuple $(D, \sqsupseteq, \sqcup, \sqcap)$, where D is a set, $\sqsupseteq$ a partial order on D, $\sqcap$ the binary *greatest lower bound operation* and $\sqcap$ the binary *least upper bound operation* on D. The relation $d_1 \sqsupseteq d_2$ is read as "d_1 *covers* d_2". Sometimes we write $d_1 \sqsubseteq d_2$ in place of $d_2 \sqsupseteq d_1$ and use $d_1 \sqsupset d_2$ (or $d_1 \sqsubset d_2$) for $d_1 \sqsupseteq d_2$ (or $d_1 \sqsubseteq d_2$, respectively) and $d_1 \neq d_2$.

We consider only *finite* lattices, i.e. those $(D, \sqsupseteq, \sqcup, \sqcap)$, where D is a finite set. In a finite lattice, there exist no infinite *ascending chain* $d_1 \sqsubset d_2 \sqsubset d_3 \cdots$. A finite lattice is always *complete*, i.e. satisfies that each subset $A \subseteq D$ has a least upper bound and greatest lower bound in D.

3.1 Constraints

A constraint-solving framework has a *constraint domain* $(\mathcal{L}, \mathcal{M}, [\![\_]\!])$ consisting of a *language* $\mathcal{L}$, a *carrier* $\mathcal{M}$ an *interpretation* $[\![\_]\!]$ (cf. e.g. [9]).

In our case, the language $\mathcal{L}$ is a first-order order-sorted language, which extends the usual first-order many-sorted language (cf. e.g. [6]) with a subsort relation (cf. e.g. [16]). As usual, $\mathcal{L}$ has a logical part consisting of usual logical constants, connectives, quantifiers, variables, and a non-logical part consisting of sorts, the subsort relation, function and predicate symbols.

A *sort* in $\mathcal{L}$ is a name composed of low-case letters. A *subsort* relation is a partial relation on sorts. For simplicity, we may not explicitly name a sort of standard composite data such as finite sets, finite mappings ord lists.

A *function* in $\mathcal{L}$ is a symbol that has exactly one *arity* of the form $\overline{a_n} \to b$ for sorts $a_1, \cdots, a_n$ and b; the function is called a *constant* if $n = 0$. A *predicate* in $\mathcal{L}$ is a symbol that has one or more *arities* of the form $\overline{a_n}$ for sorts $a_1, \cdots, a_n$. The predicates $=$ and $\neq$ exist for each sort. There is a set of variables for each sort.

Logical formulas and *terms* are built and sorted as usual. If a term has a sort, then it has each of the supersorts as its sort. Each term has a least sort.

We use $s[\overline{s_n}]$ to denote a term or logical formula containing (the occurrences of) the subterms $\overline{s_n}$. If $s[\overline{s_n}]$ and $s[\overline{t_n}]$ are in the same context, then $s[\overline{t_n}]$ is the term obtained from $s[\overline{s_n}]$ by replacing each s_i by t_i.

A variable may be *bound* (by $\forall$ and $\exists$) as usual. A variable is *free* if it is not bound. We use $\mathcal{FV}(s)$ to denote the set of all free variables in a term or a logical formula s. A term or logical formula containing no free variables is called *closed*.

We use the completely capitalized version of a sort name to denote a variable of the sort, the partially capitalized version of a sort name, where only the first letter is a capital, to range over all terms of the sort, and the name of a sort to range over all closed terms of the sort. For example, *REFS* stands for a variable of the sort *refs*, *Pp* and *Ptty* stand for a term of the sorts *pp* and *ptty*, respectively, and *refs* stands for a closed term of the sort *refs*. and *pp* stands for a closed term of the sort *pp*.

Although it is crucial to choose a particular carrier in a constraint-solving framework (see e.g. [9]), the interpretation of the terms and logical formulas we will really use is quite standard. For simplicity, we just mention that the structure $\mathcal{M}$ is an order-sorted initial algebra containing

- the sets $[\![a]\!]$ for all sorts a such that $[\![a]\!] \subseteq [\![b]\!]$ if a is a subsort of b,
- the functions $[\![f : \overline{a_n} \to b]\!] : \overline{[\![a_n]\!]} \to [\![b]\!]$ for all function symbols $f : \overline{a_n} \to b$, and
- the relations $[\![r : \overline{a_n}]\!] : \overline{[\![a_n]\!]}$ for all predicate symbols $r : \overline{a_n}$

and satisfying all axioms defining functions and predicates. For convenience we will not distinguish between a sort a (a function $f : \overline{a_n} \to b$, or a predicate $r : \overline{a_n}$) and its interpretation $[\![a]\!]$ ($[\![f : \overline{a_n} \to b]\!]$, or $[\![r : \overline{a_n}]\!]$, respectively). Furthermore, we will use closed terms and closed logical formulas to refer elements and assertions in $\mathcal{M}$.

A *constraint* is a logical formula. A set of constraints $\{\overline{s_m}\}$ represents the logical formula $s_1 \wedge \cdots \wedge s_m \wedge true$.

A *variable assignment* is a mapping $\{\overline{X_n \mapsto s_n}\}$, where each X_i is a variable and each s_i is a closed term of the sort of X_i for all $i = 1, \cdots, n$. A variable assignment as above maps each element other than $\overline{X_n}$ to itself. We use σ to range over all variable assignments. We assume that in $\sigma(s)$, bound variables are automatically renamed whenever necessary to avoid bound variable captures.

A constraint s is *satisfied under* a variable assignment σ if and only if $\sigma(s)$ is closed and holds (in $\mathcal{M}$). A constraint is *satisfiable* if and only if it is satisfied under a variable assignment.

Let X be a variable and s a closed term. Then the constraint $X = s$ is satisfiable if and only if s is an element of the sort of X.

For convenience, we may define a function f to have a result sort a and regard terms of the form $f(\overline{s_n})$ as terms of the sort a, but allow f to yield a special value *failure* not in a for some (unusual) arguments. When we write $f(\overline{s_n})$, we always implicitly mean that $f(\overline{s_n})$ should not yield *failure*. Formally, we could always have defined a new supersort a' of a, let a' contain a new constant *failure_a*, re-define the f to have the result sort a', replace each occurrence of the form $f(\overline{s_n})$ by a fresh variable X of the sort a, and add the constraint $X = f(\overline{s_n})$ in the context. The reason why the constraint $X = f(\overline{s_n})$ assures that "$f(\overline{s_n})$ is not equal to *failure_a*" is that *failure_a* is not in the sort a and thus $X = \mathit{failure\_a}$ is never satisfiable.

4 The instance method *Mth* and its static types

For the formal treatment in this paper, we consider an arbitrary but fixed instance method *Mth*, which is declared in a class *Cls*, has a method head of the form `Methodvoid` $Mth(\overline{ty_n})$, has *LocN* local variables and *InstrN* instructions in the body. The treatment of other kinds of methods is similar and thus omitted here. We assume that local variables are indexed by $0, \cdots, LocN - 1$ and program points are the integers $0, \cdots, InstrN - 1$. For the sake of uniformity, we also assume that -1 is the program point of the method head. We use *pp* to denote the sort of all program points, and *ind* the sort of all local variable indices. We use *instr* to denote the sort of all instructions.

4.1 Static types

Figure 4 formally defines *static types*, where a singleton set $\{cnam\}$ can be denoted as *cnam*. Beside the static types mentioned in Section 2, class sets are the only new thing here. A class set intuitively denotes the least common superclass of all classes in that set with respect to the subtyping relation in JVM. Since we use class sets, we need not explicitly consider the subtyping relation any more in this paper. In [13], we use reference type sets, which may also contain interfaces. The concept of reference type sets significantly simplifies the treatment, since two interfaces in JVM need not have a least superinterface.

Class set	$\{\overline{cnam_n}\} (n > 0)$	where each $cnam_i$ is a class name.
Primitive type	int	
Subroutine type	$sbr(pp)$	where each pp is a program point.
Unusable value type	$unus$	
The smallest	$\bot$	

Figure 4: Static types

We define a partial order $\sqsupseteq$ on static types as the smallest reflexive and transitive relation satisfying that

$$\begin{array}{ll} \{\overline{cnam_n}\} \sqsupseteq \{\overline{cnam_m}\} & \text{for all } n \text{ and } m \text{ with } n \geq m \\ unus \sqsupseteq stty \sqsupseteq \bot & \text{for all static types } stty \end{array}$$

Since we can define the meet and join operations $\sqcap$ and $\sqcup$ easily based on the $\sqsupseteq$ and there are only finite many static types, static types form a finite lattice.

We state some properties of the static types, which will be used later.

Lemma 1. *Assume that $sbr(pp)$ is a subroutine type and cnam an arbitrary class. Let any, any' and any'' be three static types.*

1. *For $\oslash \in \{\sqcap, \sqcup\}$, if $any \oslash any'$ is int or $sbr(pp)$ for some pp, then either $any = any \oslash any'$ or $any' = any \oslash any'$.*
2. *If $any \sqsupseteq any'$, $any' \neq \bot$ and any is int or $sbr(pp)$ for some pp, then $any = any'$.*
3. *If $any \sqsupseteq any'$, $any' \neq \bot$ and any' is not int (or not of the form $sbr(pp)$ for some pp), then any is not int (or not of the form $sbr(pp)$, respectively).*
4. *Assume that any' is int, $sbr(pp)$ or cnam, and any'' is int or $sbr(pp')$. If $any \sqsupseteq any'$, $any \sqsupseteq any''$ and $any' \neq any''$, then $any = unus$. If $any' \sqsupseteq any$, $any'' \sqsupseteq any$ and $any' \neq any''$, then $any = \bot$.*

4.2 Sorts of static types

In order to enable a simple and precise description, we use the syntax in Figure 5 to define sorts of static types, where the non-terminals on the left of ::= are sorts. Sorts of static types correspond to sets of static types. The subsort relation on these sorts is defined so that two sorts are under the subsort relation if and only if the corresponding sets are under the subset relation. For example, *ty* is a supersort of *cnam*, *stty* is a supersort of each sort of static types, and since $\{cnam\}$ and *cnam* are regarded as the same static type, *refs* is a supersort of *cnam*.

Class names	$cnam$	::=	$\cdots$ a class name $\cdots$
Types	ty	::=	$cnam \mid int$
Class sets	$refs$	::=	$\{\overline{cnam_n}\}$ $(n > 0)$
Subroutine types	sbr	::=	$sbr(pp)$
Class sets or subroutine types	$refs\_sbr$	::=	$refs \mid sbr$
Static types	$stty$	::=	$int \mid refs\_sbr \mid unus$

Figure 5: Sorts of static types

4.3 Types based on static types and variable assignments

In this section we define several sorts. Each of them contains the artificial elements $\bot$ and $\top$ indicating the beginning and a failure of the fixed-point iteration, respectively.

First, we define the sort *lvsty* of *all local variable type lists*, and the sort *stkty* of all *operand type stacks*, which are lists of the form $[stty_0, \cdots, stty_n]$with $n \geq -1$ consisting of static types.

To record local variables modified in a subroutine, we define *modification histories* as lists $[\overline{(sb_n, inds_n)}]$ of pairs of subroutines and sets of indices of local variables such that $sb_i \neq sb_j$ for all $1 \leq i \neq j \leq n$. We use *mod* to denote a sort of them. Intuitively, a modification history $[\overline{(sb_n, inds_n)}]$ represents that

- the subroutine sb_1 is called directly by the enclosing method(i.e. not by another subroutine),
- the subroutine sb_i is called by the subroutine sb_{i-1} for each i with $1 < i \leq n$,
- the set $inds_i$ consists of the indices of all local variables modified from sb_i to sb_{i+1} for each i with $1 \leq i < n$, and
- sb_n consists of the indices of all local variables modified from sb_n to the current program point.

We use the following function to add an index set *inds* to the last pair in a modification history $\overline{(sb_n, inds_n)}$

$$
\begin{array}{l}
addmlvs(inds, \overline{(sb_n, inds_n)}) := \\
\quad \begin{cases} [\overline{(sb_{n-1}, inds_{n-1})}] + (sb_n, inds_n \cup inds) & \text{if } n \geq 1 \\ [] & \text{otherwise} \end{cases}
\end{array}
$$

Note that in the full JVM, a modification history needs to be expressed as a directed acyclic graph, where a node stands for a subroutine and a directed edge for a call to the subroutine represented by the ending node by the subroutine represented by the beginning node, since more than one subroutine may call the same subroutine and one subroutine may call more than one subroutine. Although there are no fundamental problems at all to do this, the treatment appears too tedious to be described in this paper. Thus we choose to consider the simplified case, where each program point of a well-typed program can only have a single list of called (and not yet completed) subroutines.

Now we define *program point types* as triples $(lvsty, stkty, mod)$ consisting of local variable type lists, operand type stacks and modification histories and use *ptty* denote the sort of them.

Finally we define *program types* (of the method *Mth*) as mappings

$$\{pp \mapsto ptty_{pp} \mid \text{for each (program point) } pp \geq 0\}$$

from program points to program point types and use *prgty* denote the sort of them.

For each sort defined in this section, we define a partial relation $\sqsupseteq$, the meet operation $\sqcap$ and the join operation $\sqcup$ satisfying the following:

- $\bot$ is the smallest and $\top$ the biggest element with respect to $\sqsupseteq$ in the sort,
- All elements that contain an occurrence of $\bot$, including the static type $\bot$, are regarded as identical to $\bot$. All elements that contain an occurrence of $\top$ are regarded as identical to $\top$. Note that the biggest static type *unus* does not indicate a failure.

- The equations in Figure 6 hold for each $\oslash \in \{\sqcap, \sqcup\}$, where we write $\odot_{\sqcap}$ for $\cap$, $\odot_{\sqcup}$ for $\cup$, $\mathbb{I}_{\sqcap}$ for $\perp$ and $\mathbb{I}_{\sqcup}$ for $\top$.

These sorts are all finite lattices.

- For $listy = [stty_0, \cdots, stty_n]$ and $listy' = [stty'_0, \cdots, stty'_m]$,

$$
\begin{array}{l}
listy \sqsupseteq listy' \iff n = m \text{ and } stty_i \sqsupseteq stty'_i \text{ for all } i = 0, \ldots, n \\
listy \oslash listy' = \begin{cases} [stty_0 \oslash stty'_0, \cdots, stty_n \oslash stty'_n] & \text{if } m = n \\ \mathbb{I}_{\oslash} & \text{otherwise} \end{cases}
\end{array}
$$

- For $mod = [\overline{(sb_n, inds_n)}]$ and $mod' = [\overline{(sb_m, inds'_m)}]$,

$$
\begin{array}{l}
mod \sqsupseteq mod' \iff n = m \text{ and } inds_i \supseteq inds'_i \text{ for all } i = 0, \ldots, n \\
mod \oslash mod' = \begin{cases} [\overline{(sb_n, inds \odot_{\oslash} inds'_n)}] & \text{if } m = n \\ \mathbb{I}_{\oslash} & \text{otherwise} \end{cases}
\end{array}
$$

- For $ptty = (lvsty, stkty, mod)$ and $ptty' = (lvsty', stkty', mod')$,

$$
\begin{array}{l}
ptty \sqsupseteq ptty' \iff lvsty \sqsupseteq lvsty',\ stkty \sqsupseteq stkty' \text{ and } mod \sqsupseteq mod' \\
ptty \oslash ptty' := (lvsty \oslash lvsty', stkty \oslash stkty', mod \oslash mod')
\end{array}
$$

- For $prgty$ and $prgty'$,

$$
\begin{array}{l}
prgty \sqsupseteq prgty' \iff prgty(pp) \sqsupseteq prgty'(pp) \text{ hold for all } pp \\
prgty \oslash prgty' := \{pp \mapsto prgty(pp) \oslash prgty'(pp) \mid \text{for each } pp\}
\end{array}
$$

Figure 6: Definitions of $\sqsupseteq$, $\sqcap$ and $\sqcup$ for the sorts *lvsty*, *stkty*, *mod*, *ptty* and *prgty*

We say that the sort of a variable X is *based on static types* if and only if the sort is *stty*, a subsort of *stty*, *lvsty*, *stkty*, *mod*, *ptty* or *prgty*. For two variable assignments σ_1 and σ_2 with $\mathcal{D}om(\sigma_1) = \mathcal{D}om(\sigma_2)$, we define the relation $\sigma_1 \sqsupseteq \sigma_2$ by that $\sigma_1 \sqsupseteq \sigma_2$ holds if and only if for all $X \in \mathcal{D}om(\sigma_1)$,

$$
\begin{array}{ll}
\sigma_1(X) \sqsupseteq \sigma_2(X) & \text{if the sort of } X \text{ is based on static types} \\
\sigma_1(X) = \sigma_2(X) & \text{otherwise}
\end{array}
$$

For σ_1 and σ_2 such that $\mathcal{D}om(\sigma_1) = \mathcal{D}om(\sigma_2)$ and $\sigma_1(X) = \sigma_2(X)$ hold for all $X \in \mathcal{D}om(\sigma_1)$ such that the sort of X is not based on static types, we define $\sigma_1 \oslash \sigma_2$ by that $\mathcal{D}om(\sigma_1 \oslash \sigma_2) = \mathcal{D}om(\sigma_1)$ and

$$
\sigma_1 \oslash \sigma_2(X) = \begin{cases} \sigma_1(X) \oslash \sigma_2(X) & \text{if the sort of } X \text{ is based on static types} \\ \sigma_1(X) & \text{otherwise} \end{cases}
$$

5 The form of typing rules and well-typedness

From now on we also write P and SB as variables of the sort *pp*, LT, ST, M, Π and Φ as variables of the sorts *lvsty*, *stkty*, *mod*, *ptty* and *prgty*, respectively, for notational simplicity.

The following two forms of constraints are particularly important:

$$Prgty(Pp) = Ptty \quad \text{and} \quad Prgty(Pp) \sqsupseteq Ptty$$

The former says that the program point type at Pp in $Prgty$ should be $Ptty$, and the latter says that it should cover $Ptty$. If a program point Pp can be reached by more than one preceding program point, then it is quite convenient to use the latter form to constrain the program point type at Pp.

In general, a typing rule is in the form:

$$\frac{\boxed{\mathcal{AC}} \\ \mathcal{CC}}{\mathcal{SC}}$$

The $\mathcal{AC}$, $\mathcal{CC}$ and $\mathcal{SC}$ are sets of constraints. The constraints in $\mathcal{AC}$ are called *applicability conditions*, among which a distinguished constraint of the form $Mth(P) = Instr$ exists. Intuitively, $\mathcal{AC}$ is used to determine a program point P, where the typing rule should be applied. In particular, $Mth(P) = Instr$ says that the instruction at a program point P in Mth should be of the form $Instr$. The $\mathcal{CC}$ contains constraints on the program point type at the current program point P. The $\mathcal{SC}$ consists of zero or more logical formulas of the form $\Phi(Pp) \sqsupseteq Ptty$, which are constraints on the program point types at successor program pointsPp of the instruction.

Intuitively, all typing rules enforce constraints on one common program type. Therefore, all typing rules contain one common variable Φ for a program type.

Syntactically, a typing rule always satisfies the following conditions:

$$\mathcal{FV}(\mathcal{AC} \cup \mathcal{CC}) \cup \{\Phi\} \supseteq \mathcal{FV}(\mathcal{SC})$$
$$\mathcal{FV}(\mathcal{AC}) \supseteq \mathcal{FV}(Pp) \text{ for each } \Phi(Pp) \sqsupseteq Ptty \in \mathcal{SC}$$

Let Q denote $\mathcal{FV}(\mathcal{AC}) - \{\Phi\}$ and Q' denote $\mathcal{FV}(\mathcal{CC} \cup \mathcal{SC}) - (\{\Phi\} \cup Q)$. Then a typing rule formally denotes the constraint

$$\forall Q.(\mathcal{AC} \Rightarrow \exists Q'.(\mathcal{CC} \cup \mathcal{SC}))$$

It is easy to see that the constraint is satisfied under a variable assignment $\{\Phi \mapsto prgty\}$ if and only if, if $\mathcal{AC}$ is satisfied under a variable assignment σ with $\mathcal{D}om(\sigma) = Q \cup \{\Phi\}$ and $\sigma(\Phi) = prgty$, then there is a variable assignment σ' with $\mathcal{D}om(\sigma) = Q \cup Q' \cup \{\Phi\}$ such that $\sigma'_{|Q \cup \{\Phi\}} = \sigma$ and $\sigma'(\mathcal{CC} \cup \mathcal{SC})$ is satisfied.

The reason for us to separate the sets $\mathcal{CC}$ and $\mathcal{SC}$ in a typing rule will become clear in Section 7.

We say that *Mth has a program type prgty*, (or *prgty is a program type* of *Mth*,) if and only if the constraints denoted by all typing rules are satisfied under $\{\Phi \mapsto prgty\}$. The method Mth may have zero, one or more than one program type.

A program is *legal* (or *statically well-typed*) if and only if it has a program type.

6 The formal specification

This section gives the typing rules. In the rules we use _ to denote a wildcard variable. We assume the following:

- The $\mathcal{AC}$ part of each typing rule except rule (T-7) always implicitly contains the constraint $\Phi(P) \neq \perp$.
- Each constraint of the form $\Phi(Pp) = Ptty$ in the $\mathcal{AC}$- or $\mathcal{CC}$-part of a typing rule always implicitly stands for $\Phi(Pp) \neq \perp \,\wedge\, \Phi(Pp) = Ptty$.
- The $\mathcal{CC}$ in a typing rule always implicitly contains a constraint $Pp = PP$ (or $Pp \leq InstrN$) for each $\Phi(Pp) \sqsupseteq Ptty$ in the $\mathcal{SC}$, where PP is a fresh variable of the sort pp. The constraint ensures that Pp is a program point.

The typing rules for some instructions are given Figure 7. We explain rule (T-1) to show some of the tricky points in the formulation of a typing rule. Similar explanations can be given for other typing rules.

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{aload}\ IND}\\ \Phi(P) = \Pi[LT, ST]\\ REFS = LT(IND)\end{array}}{\Phi(P+1) \sqsupseteq \Pi[LT, ST + REFS]} \tag{T-1}$$

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{iload}\ IND}\\ \Phi(P) = \Pi[LT, ST]\\ int = LT(IND)\end{array}}{\Phi(P+1) \sqsupseteq \Pi[LT, ST + int]} \tag{T-2}$$

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{astore}\ IND}\\ \Phi(P) = \Pi[LT, ST + REFS\_SBR, M]\end{array}}{\Phi(P+1) \sqsupseteq \Pi[LT[n \mapsto REFS\_SBR], ST, addmlvs(\{IND\}, M)]} \tag{T-3}$$

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{istore}\ IND}\\ \Phi(P) = \Pi[LT, ST + int, M]\end{array}}{\Phi(P+1) \sqsupseteq \Pi[LT[n \mapsto int], ST, addmlvs(\{IND\}, M)]} \tag{T-4}$$

Figure 7: Typing rules for load and store instructions

First, $REFS = LT(IND)$ in rule (T-1) intuitively expresses a membership constraint, i.e. that $LT(IND)$ should be in the sort *refs*, since $REFS$ is a variable of the sort *refs*. It implies that an `aload` must load objects. In addition, rule (T-1) says that the local variable type list, the operand type stack and the modification history at $P+1$ should cover the corresponding component at P, except that the operand type stack at $P+1$ should be extended by a static type covering $LT(IND)$. Note that the variables Φ and

LT in $\Phi(P)$ and $LT(IND)$ are not higher-order variables, since $\Phi(P)$ is in fact an application of an implicit function *app* on two first-order variables Φ and P, and $LT(IND)$ that on LT and IND.

Similar explanations can be given for other typing rules. Note that rules (T-3) and (T-4) affect modification histories. More concretely, the term $addmlvs(\{IND\}, M)$ in these rules adds the index IND of the modified local variable into the modification history M.

It is worth noticing that in rule (T-3) the variable $REFS\_SBR$ can be instantiated into an element of the sort *sbr*, whereas the variable $REFS$ in rule (T-1) cannot. This means that an `astore` instruction can store a program point, whereas an `aload` instruction cannot load it.

$$\frac{\boxed{Mth(P) = \texttt{if\_acmpeq}\ N} \quad \Phi(P) = \Pi[ST + REFS + REFS']}{\Phi(N) \sqsupseteq \Pi[ST] \quad \Phi(P+1) \sqsupseteq \Pi[ST]} \tag{T-5}$$

$$\frac{\boxed{Mth(P) = \texttt{goto}\ N} \quad \Phi(P) = \Pi}{\Phi(N) \sqsupseteq \Pi} \tag{T-6}$$

Figure 8: Typing rules for control transfer instructions

The typing rules for the control transfer instructions in Figure 8 need no further explanations.

$$\frac{\boxed{\begin{array}{l} Mth(P) = \texttt{Method void}\ Mth(\overline{TY_n}) \\ n \leq LocN \end{array}}}{\Phi(P+1) \sqsupseteq (unus^{LocN}[0 \mapsto Cls, \overline{n \mapsto TY_n}], [], \emptyset)} \tag{T-7}$$

$$\frac{\boxed{Mth(P) = \texttt{return}}}{} \tag{T-8}$$

Figure 9: Typing rule for the starting program point of a method code

The typing rules for starting and terminating the method *Mth* are defined in Figure 9. Rule (T-7) treats the implicit instruction starting a method. It says that at the beginning of a method, if a local variable does not store the object on which the method is invoked, nor an actual parameter, then it is given the type *unus*, ensuring that its content is unusable before a value is explicitly assigned to it. We use $unus^m$ to denote the list $[unus, \cdots, unus]$

consisting of m times *unus*. Note that the modification history at the beginning of a method is $\emptyset$. Rule (T-8) means that no explicit constraints on the program point type at a `return` are needed (in the context of the current paper).

The typing rules for `jsr` and `ret` are given in Figure 10. Rule (T-9) uses $SB \notin \mathcal{D}om(M)$ to assure that the subroutine SB is not called recursively. Furthermore, it ensures that the modification history at the beginning of the subroutine SB extends the one at the call site by a pair $(SB, \emptyset)$, indicating that the subroutine SB is called.

$$\frac{\boxed{Mth(P) = \texttt{jsr}\ SB} \quad \Phi(P) = \Pi[ST, M] \quad SB \notin \mathcal{D}om(M)}{\Phi(SB) \sqsupseteq \Pi[ST + sbr(SB), M + (SB, \emptyset)]} \qquad \text{(T-9)}$$

$$\frac{\begin{array}{l} \boxed{Mth(P) = \texttt{ret}\ IND} \\ \Phi(P) = \Pi[LT] \\ LT(IND) = sbr(\_) \\ \forall P' \forall IND' \forall \Pi' \forall LT'.(\ (Mth(P') = \texttt{ret}\ IND' \ \wedge\ P' \neq P \ \wedge\ \Phi(P') = \Pi'[LT']) \\ \qquad\qquad\qquad \Rightarrow LT(IND) \neq LT'(IND')\ ) \end{array}}{} \qquad \text{(T-10)}$$

$$\frac{\boxed{\begin{array}{l} Mth(P) = \texttt{ret}\ IND \\ \Phi(P) = (LT, ST, M) \\ Mth(P') = \texttt{jsr}\ SB \\ LT(IND) = sbr(SB) \\ \Phi(P') = (LT', ST', M') \end{array}}}{\begin{array}{l} \Phi(P'+1) \sqsupseteq (LT'[j \mapsto turnUnus(LT(j), sbsin(M,SB)) \mid j \in mlvsin(M,SB)], \\ \qquad turnUnus(ST, sbsin(M, SB)), \\ \qquad addmlvs(mlvsin(M, SB), modtill(M, SB)) \end{array}} \qquad \text{(T-11)}$$

Figure 10: Typing rules for `ret` and `jsr`

Rules (T-10) and (T-11) are both for `ret`. We could have combine them into one, but the resulting typing rule would become too clumsy.

In rule (T-10), the constraint $LT(IND) = sbr(\_)$ assures that the local variable IND holds a program point. The constraint $\forall P' \forall IND' \forall \Pi' \forall LT'. \cdots$ assures that Mth has at most one `ret` for the same subroutine. This is not a serious restriction due to the presence of `goto`.

In rule (T-11), the applicability conditions assure that SB is a subroutine, the `jsr` at P' is one of `jsr`'s that call the subroutine, the `ret` at P terminates the subroutine and causes control to return to $P'+1$. Note that the constraint

$\forall P' \forall IND' \forall \Pi' \forall LT'. \cdots$ in rule (T-10) assures that in rule (T-11), the `ret` at P is the unique `ret` for the subroutine SB.

The formulation of rule (T-11) uses several auxiliary functions. Let $mod = [\overline{(sb_n, inds_n)}]$. Then the first group of auxiliary functions are defined as follows:

$$sbsin(mod, sb) := \begin{cases} \{sb_k, \cdots, sb_n\} & \text{if } sb = sb_k \text{ for } 1 \leq k \leq n \\ failure & \text{otherwise} \end{cases}$$
$$mlvsin(mod, sb) := \begin{cases} \bigcup_{k \leq i \leq n} inds_i & \text{if } sb = sb_k \text{ for } 1 \leq k \leq n \\ failure & \text{otherwise} \end{cases}$$
$$modtill(mod, sb) := \begin{cases} [\overline{(sb_{k-1}, inds_{k-1})}] & \text{if } sb = sb_k \text{ for } 1 \leq k \leq n \\ failure & \text{otherwise} \end{cases}$$

Assume that the program point type at a program point pp contains the modification history mod. Then the function $sbsin(mod, sb)$ with $sb \in \mathcal{D}om(mod)$ intuitively computes a set containing all subroutines called from sb to pp. The function $mlvsin(mod, sb)$ with $sb \in \mathcal{D}om(mod)$ computes the set of the indices of local variables that may be modified from sb to pp. The function $modtill(mod, sb)$ computes the result of mod obtained by cutting all subroutines after (and including) the subroutine sb.

In order to change all subroutine types of the subroutines in a set E into the static type $unus$, we define the following auxiliary functions $turnUnus$ and $turnUnusLis$:

$$turnUnus(stty, E) := \begin{cases} unus & \text{if } stty = sbr(sb) \text{ and } sb \in E \\ stty & \text{otherwise} \end{cases}$$
$$turnUnusLis([\overline{stty_m}], E) := [\overline{turnUnus(stty_m, E)}]$$

Lemma 2. *Let stty be a static type. Then* $turnUnus(stty, E) \sqsupseteq stty$ *always holds.*

Proof. Follows easily from the definition. □

Rule (T-11) enforces the following constraints on the program point type at the return program point $P' + 1$:

- The term $LT'[j \mapsto turnUnus(LT(j), sbsin(M, SB)) \mid j \in mlvsin(M, SB)]$ intuitively means that if a local variable j is not modified from SB to P (i.e. not in $mlvsin(M, SB)$), then its static type at $P' + 1$ should cover $LT'(j)$ (at P'); otherwise it shoudl cover $LT(j)$ (at P), where if $LT(j)$ is a subroutine type for a subroutine called from SB to P (i.e. in $sbsin(M, SB)$), then the static type for the local variable j at $P' + 1$ should be $unus$.
- The term $turnUnus(ST, sbsin(M, SB))$ means that the operand stack type at $P' + 1$ should cover ST (at P), where if an operand stack entry in ST has a subroutine type for a subroutine called from SB to P, then the corresponding static type at $P' + 1$ should be $unus$.

– The term $addmlvs(mlvsin(M, SB), modtill(M, SB))$ intuitively means that the list of modified variables at $P' + 1$ should be obtained from the one at P by cutting the subroutine SB and all subroutines called by it, and all local variables modified in the subroutine SB, including those modified in subroutines called by it, should be regarded as being modified at $P' + 1$.

7 Computing a program type or yielding a failure

The typing rules given in Section 6 formally define all program types of the method *Mth*: a program type is one program type of *Mth* if and only if it satisfies all constraints defined by these rules. Although one could generate all possible program types and check whether they satisfy the constraints one by one, it is definitely desirable to have a constraint-solving algorithm to compute a program type. In addition, it is interesting to ask whether there is a canonical (or principal) one among all program types of *Mth*.

In this section we describe an algorithm, called *analyzer*, which computes the smallest program type of the method *Mth* as the canonical one if there is any, or yields a failure, if there is none.

7.1 Using abstract interpretation

Abstract interpretation is a framework that has been successfully used in formalizing a wide variety of data-flow analysis. Since the classic presentation of data-flow analysis (in e.g. [1]) is not formal enough for our purpose in this paper, we follow the idea of abstract interpretation.

More concretely, we follow the Cousots' approach to abstract interpretation ([4], see also e.g. [11]), which aims at effectively finding safe approximations for states of memory locations at program points. In our case, a program state contains a state of local memory locations at each program point, and a program point type can serve as a safe approximation of all possible states of memory locations at a program point.

7.2 Applicability, pre-satisfaction and satisfaction of typing rules

From now on, when discussing a particular typing rule, we will use the general notations given at the beginning of Section 5 and the special notations in the formulation of the typing rule without explicitly saying.

For each typing rule, if there is a variable assignment σ with

$$\mathcal{D}om(\sigma) = \mathcal{FV}(\mathcal{AC}) \cup \{\Phi\}$$

such that $\mathcal{AC}$ is satisfied under σ, then the typing rule is said to be *applicable under* σ.

For each typing rule, if there is a variable assignment σ with

$$\mathcal{D}om(\sigma) = \mathcal{FV}(\mathcal{AC} \cup \mathcal{CC}) \cup \{\Phi\}$$

such that $\mathcal{AC} \cup \mathcal{CC}$ is satisfied under σ, then the typing rule is said to be *pre-satisfied under* σ.

If a typing rule is pre-satisfied under σ and $\mathcal{SC}$ is satisfied under σ, then it is said to be *satisfied under* σ.

We also say that a typing rule is *applicable* (*pre-satisfied* or *satisfied*) *at the program point* $\sigma(P)$ or *with respect to the program type* $\sigma(\Phi)$ if it is applicable (pre-satisfied or satisfied) under σ.

7.3 The algorithm

We formulate the algorithm *analyzer* in Figure 11, which produces a sequence of *intermediate program types* $prgty_0, prgty_2, \cdots$, and finally terminates with either a program type of the method *Mth* or a failure. The algorithm uses a boolean function *monoton_assure* to slightly restrict the flexibility of the application of rule (T-11) and calls a procedure *apply_a_rule* to compute each intermediate program type.

Input: The method *Mth*.
Output: A program type or failure.
Body:

1. Let $prgty_0 := \bot$.
2. Assume that $prgty_k$ is the last obtained intermediate program type. Choose a typing rule and a variable assignment σ such that $\sigma(\Phi) = prgty_k$ holds, the rule is applicable under σ, and if the rule is rule (T-11) then *monoton_assure*(σ) yields *true*. Let *apply_a_rule* take the typing rule and σ as arguments and run.
   (a) If *apply_a_rule* yields a failure, then the algorithm terminates with a failure;
   (b) otherwise, if *apply_a_rule* yields a program type different than $prgty_k$, then let the yielded program type be a new intermediate program type $prgty_{k+1}$.
3. If the above step yields a new intermediate program type $prgty_{k+1}$, then repeat the step with $k := k + 1$, until the algorithm terminates with a failure, or *apply_a_rule* does not yield a new intermediate program type for all typing rules and variable assignments chosen as above. In the latter case, the algorithm terminates with the last computed intermediate program type.

Figure 11: The algorithm *analyzer*

The procedure *apply_a_rule* is defined in Figure 12. If a typing rule is given, then the procedure becomes a transfer function based on that rule. The transfer function intuitively takes the last obtained intermediate program type $prgty_k$ as argument and attempts to compute a new one $prgty_{k+1}$. The $prgty_{k+1}$ should be bigger than $prgty_k$ and not equal to $\top$, satisfy the constraints enforced by the typing rule, thus be closer than $prgty_k$ to a program type of *Mth*, if it exists. Technically, *apply_a_rule* takes a variable assignment σ, instead of a single program type, as argument, since the $\mathcal{AC}$ part of a typ-

Input: A typing rule and a variable assignment σ such that the typing rule is applicable under σ.
Output: A program type or failure.
Body:

1. If $find\_a\_subst$ yields a variable assignment σ' for the typing rule and σ such that $\sigma'(\Phi(Pp)) \sqcup \sigma'(Ptty) \neq \top$ for all $\Phi(Pp) \sqsupseteq Ptty \in \mathcal{SC}$, then the current procedure yields the program type

$$\sigma'(\Phi)[\sigma'(Pp) \mapsto \sigma'(\Phi(Pp)) \sqcup \sigma'(Ptty) \mid \Phi(Pp) \sqsupseteq Ptty \in \mathcal{SC}]$$

2. otherwise, the current procedure yields a failure.

Figure 12: The procedure *apply_a_rule*

ing rule may contain other variables than Φ, and a variable assignment also determines the assignment of these other variables.

Input: A typing rule and a variable assignment σ such that the typing rule is applicable under σ.
Output: A variable assignment or failure.
Body:

1. Construct all variable assignments σ' such that $\sigma'_{|\mathcal{FV}(\mathcal{AC}) \cup \{\Phi\}} \sqsupseteq \sigma$ holds and the typing rule is pre-satisfied under σ'.
2. If no such variable assignments exist, then yield a failure;
3. Otherwise, yield the result of applying the operation $\sqcap$ to all these variable assignments.

Figure 13: The procedure $find\_a\_subst$

The procedure *apply_a_rule* calls the procedure $find\_a\_subst$ in Figure 13. The procedure $find\_a\_subst$ computes a variable assignment σ' such that $\sigma'(\Phi) \sqsupseteq prgty$ and the typing rule is pre-satisfied under σ', or reports a failure. Note that the procedure $find\_a\_subst$, as defined in Figure 13, is naive and inefficient. We could have written a more efficient one, but the proof would become complicated.

The boolean function *monoton_assure* is defined in Figure 14. It means that if a local variable j is not modified from $\sigma(P')$ to $\sigma(P)$ (according to the modification history *mod* in $\sigma(\Phi(P))$ at $\sigma(P)$), then the static type of the local variable j at $\sigma(P)$ covers that at $\sigma(P')$. It formalizes the additional applicability condition mentioned in Section 2.3.

7.4 Fixed points

A program type *prgty* is called a *fixed point restricted to a typing rule and a variable assignment* σ if and only if whenever the typing rule is applicable

Input: A variable assignment σ under which rule (T-11) is applicable.
Output: *true* or *false*.
Body: We use the notations in rule (T-11). Let $\sigma(\Phi(P)) = (lvsty, \cdots, mod)$ and $\sigma(\Phi(P')) = (lvsty', \cdots)$. If $\sigma(SB) \in \mathcal{D}om(mod)$ and

$$\forall 0 \leq j < LocN. j \notin mlvsin(\sigma(SB), mod) \Rightarrow lvsty(j) \sqsupseteq lvsty'(j)$$

holds, then the current function yields *true*; otherwise the function yields *false*.

Figure 14: The boolean function *monoton_assure*

under σ and $\sigma(\Phi) = prgty$ holds, there is a variable assignment σ' such that the typing rule is satisfied under σ' and $\sigma'_{|\mathcal{FV}(\mathcal{AC})\cup\{\Phi\}} = \sigma$ holds. A program type *prgty* is called a *fixed point* if and only if, it is a fixed point restricted to each typing rule and each variable assignment.

Intuitively, if *analyzer* terminates with a program type, then the program type must be a fixed point.

7.5 Example

Figure 15 describes how to compute the program type in Figure 2 using the algorithm *analyzer*.

Each row in Figure 15 records a program point type in an intermediate program type, together with information about the next application of a typing rule. The informnation includes a number indicating the order of the application in the column "Step", the name of the applied typing rule in the column "Rule" and all static successor program points according the typing rule in the column "Succ.'s". According to the algorithm *analyzer*, an application includes a check of the $\mathcal{CC}$-part and a computation of a new program point type at each successor program point. If more than one program point type has been computed at one program point, then from the second one on, only the changed parts are explicitly recorded in Figure 15. Note that the last program point types for all program points are exactly those in Figure 2.

The computation has 30 application steps. Step 1 applies rule (T-7) and computes a local variable type list $[C, int, int, unus, unus]$ and an operand type stack [] at the program point `0`. Step 2 applies rule (T-9), checks the local variable type list $[C, int, int, unus, unus]$ and the operand stack type [] at the program point `0` and computes a local variable type list $[C, int, int, unus, unus]$ and an operand stack type $[sbr(5)]$ at the program point `5`, and so on.

After step 9, which applies rule (T-10) at `13`, rule (T-11) could be applied at `13`. But we choose first to continue at `9` to consider the second branche of **`if_acmeq`**. (Note that a static analysis needs to consider both branches of **`if_acmeq`**.) After step 14, we choose to apply rule (T-11) at `13` as step 15.

P	LT	ST	M	Step	Rule	Succ.'s
`Method void m(int,int)`				1	(T-7)	0
0	$[C,int,int,unus,unus]$	$[]$	$[]$	2	(T-9)	5
1	$[C,unus,int,unus,unus]$	$[]$	$[]$	16	(T-1)	2
2	$[C,unus,int,unus,unus]$	$[C]$	$[]$	17	(T-3)	3
3	$[C,unus,C,unus,unus]$	$[]$	$[]$	18	(T-9)	5
4	$[C,unus,C,unus,unus]$	$[]$	$[]$	30	(T-8)	returns
5	$[C,int,int,unus,unus]$	$[sbr(5)]$	$[(5,\{\})]$	3	(T-3)	6
	$[C,unus,unus,unus,unus]$			19	(T-3)	6
6	$[C,int,int,sbr(5),unus]$	$[]$	$[(5,\{3\})]$	4	(T-1)	7
	$[C,unus,unus,sbr(5),unus]$			20	(T-1)	7
7	$[C,int,int,sbr(5),unus]$	$[C]$	$[(5,\{0,3\})]$	5	(T-1)	8
	$[C,unus,unus,sbr(5),unus]$			21	(T-1)	8
8	$[C,int,int,sbr(5),unus]$	$[C,C]$	$[(5,\{0,3\})]$	6	(T-5)	11,9
	$[C,unus,unus,sbr(5),unus]$			22	(T-5)	11,9
9	$[C,int,int,sbr(5),unus]$	$[]$	$[(5,\{0,3\})]$	10	(T-1)	10
	$[C,unus,unus,sbr(5),unus]$			23	(T-1)	10
10	$[C,int,int,sbr(5),unus]$	$[C]$	$[(5,\{0,3\})]$	11	(T-3)	11
	$[C,unus,unus,sbr(5),unus]$			24	(T-3)	11
11	$[C,int,int,sbr(5),unus]$	$[]$	$[(5,\{0,3\})]$	7	(T-9)	12
	$[C,unus,int,sbr(5),unus]$		$[(5,\{0,1,3\})]$	12	(T-9)	12
	$[C,unus,unus,sbr(5),unus]$			25	(T-9)	12
12	$[C,int,int,sbr(5),unus]$	$[sbr(12)]$	$[(5,\{0,3\}),(12,\{\})]$	8	(T-3)	13
	$[C,unus,int,sbr(5),unus]$		$[5{\mapsto}\{0,1,3\},12{\mapsto}\{\}]$	13	(T-3)	13
	$[C,unus,unus,sbr(5),unus]$			26	(T-3)	13
13	$[C,int,int,sbr(5),sbr(12)]$	$[]$	$[(5,\{0,3\}),(12,\{4\})]$	9	(T-10)	
	$[C,unus,int,sbr(5),sbr(12)]$		$[(5,\{0,1,3\}),(12,\{4\})]$	14	(T-10)	
				15	(T-11)	1
	$[C,unus,unus,sbr(5),sbr(12)]$			27	(T-10)	
				28	(T-11)	4
				29	(T-11)	1

Figure 15: Computing the program type in Figure 1

Let mod denote $[(5, \{0,1,3\}), (12, \{4\})]$. Then we have that

$$\begin{aligned}
&mlvsin(mod, 5) = \{0, 1, 3, 4\}\\
&sbsin(mod, 5) = \{5, 12\}\\
&modtill(mod, 5) = []
\end{aligned}$$

Since the local variable 2 has the static type int at both `13` and `0`, the function $monoton\_assure$ yields $true$. Thus the application is possible. Step 15 computes a program point type at `1`. The local variables 3 and 4 at `1` have the static type $unus$, since they have the subroutine types $sbr(5)$ and $sbr(12)$ at `13`, where $3, 4 \in mlvsin(mod, 5)$ and $5, 12 \in sbsin(mod, 5)$ hold. The local variables 0 and 1 at `1` have the static types C and $unus$, respectively, covering those at `13`, since $0, 1 \in mlvsin(mod, 5)$ holds. The local variable 2 at `1` has its static type int at `0`, since $2 \notin mlvsin(mod, 5)$ holds. Note that the static type of the local variable 2 at `13` is also int. But it affects only the result yielded by the boolean function $monoton\_assure$, not directly the static type of the local variable 2 at `1`.

Step 28 applies rule (T-11) at `13` with respect to the `jsr` at `3`, computing a new program point type at `4`. The program point type at `13` contains the same mod as above, and thus the functions $mlvsin(mod, 5)$, $sbsin(mod, 5)$ and $modtill(mod, 5)$ yield the same results as above. Since now the local variable 2 at `13` has the static type $unus$ and that at `3` the static type C, $monoton\_assure$ yields $true$. Thus the application is possible. For the same reason as before, the local variables 3 and 4 at `4` have the static type $unus$, the local variables 0 and 1 at `4` have the static types C and $unus$, respectively, covering those at `13`, and in particular, the local variable 2 at `4` takes its static type C at `3`, since $2 \notin mlvsin(mod, 5)$ holds. Note that the local variable 2 at `13` has the static type $unus$: it is easy to see that it does not directly affect the static type of the local variable 2 at `4`.

8 Properties of the algorithm *analyzer*

8.1 Basic properties

Lemma 3. *If a typing rule is satisfied under σ, then the typing rule is pre-satisfied under σ. If a typing rule is pre-satisfied under σ, then the typing rule is applicable under $\sigma_{|\mathcal{FV}(\mathcal{AC}) \cup \{\Phi\}}$.*

Proof. Follows directly from the definitions. □

Each typing rule has the property that if it is applicable or pre-satisfied under a variable assignment σ, then the entire variable assignment σ can be uniquely determined by the variable assignment restricted to $\mathcal{FV}(\mathcal{AC}) \cup \{\Phi\}$.

Lemma 4. *For each typing rule and any variable assignments σ_1 and σ_2, under which the typing rule is applicable or pre-satisfied, if $\sigma_1(X) = \sigma_2(X)$ for all $X \in \mathcal{FV}(\mathcal{AC}) \cup \{\Phi\}$, then $\sigma_1 = \sigma_2$ holds.*

Proof. It is straightforward to check the following:

1. If the typing rule is not (T-11), then $\sigma_1(P) = \sigma_2(P)$ and $\sigma_1(\Phi) = \sigma_2(\Phi)$ imply that $\sigma_1 = \sigma_2$.
2. If the typing rule is rule (T-11), then $\sigma_1(P) = \sigma_2(P)$, $\sigma_1(\Phi) = \sigma_2(\Phi)$ and $\sigma_1(P') = \sigma_2(P')$ imply that $\sigma_1 = \sigma_2$.

□

Each typing rule has the property that the relation $\sqsupseteq$ on those variable assignments σ, under which the rule is applicable or pre-satisfied, is determined by the relation $\sqsupseteq$ on $\sigma(\Phi)$.

Lemma 5. *For each typing rule and any variable assignments σ_1 and σ_2, under which the typing rule is applicable or pre-satisfied, $\sigma_1 \sqsupseteq \sigma_2$ holds if and only if $\sigma_1(\Phi) \sqsupseteq \sigma_2(\Phi)$ holds, and $\sigma_1 \sqsupset \sigma_2$ holds if and only if $\sigma_1(\Phi) \sqsupset \sigma_2(\Phi)$ holds.*

Proof. The proof follows by an easy examination of each typing rule. □

It would be desirable to have the property that if a typing rule is applicable at a program point *pp* with respect to a program type *prgty*, then it remains applicable at *pp* with respect to every program type *prgty'* with *prgty'* $\sqsupseteq$ *prgty*. In reality we have the property for all typing rules except rule (T-11).

Lemma 6. *Assume that a typing rule is applicable under a variable assignment σ. Let prgty' be an arbitrary program type with prgty' $\sqsupseteq \sigma(\Phi)$. Then the following always hold:*

1. *If the typing rule is not rule (T-11), then there is always a variable assignment σ' with $\sigma'(\Phi) =$ prgty' and $\sigma' \sqsupseteq \sigma$ such that the typing rule is applicable under σ'.*
2. *If the typing rule is rule (T-11), then there is always a variable assignment σ' with $\sigma'(\Phi) =$ prgty' and $\sigma' \sqsupseteq \sigma$ such that rule (T-11) is applicable under σ' or rule (T-10) is applicable but not pre-satisfied at a program point with respect to prgty'*

Proof. The proof of 1 follows by an easy examination of each typing rule. The proof of 2 is slightly more complex, since the $\mathcal{AC}$ part of rule (T-11) requires the existence of a `ret` that uses a local variable with a corresponding subroutine type. This condition need not remain to hold if the program type becomes bigger. However, if this condition no longer holds, then (T-10) is no longer pre-satisfied at the program point of the `ret`. □

If rule (T-10) is applicable but not pre-satisfied with respect to a program type in an ascending chain of program types, then it is applicable but not pre-satisfied with respect to each program type after the program type.

Lemma 7. *If rule (T-10) is applicable but not pre-satisfied with respect to a program type $prgty_1$, and if $prgty_2$ is a program type with $prgty_2 \sqsupseteq prgty_1$, then rule (T-10) is applicable but not pre-satisfied with respect to $prgty_2$.*

Proof. Assume that rule (T-10) is applicable but not pre-satisfied at pp with respect to $prgty_1$. Then $prgty_1(pp) \neq \bot$ holds, and there is a variable assignment σ_1 such that $\sigma_1(P) = pp$, $\sigma_1(\Phi) = prgty_1$, $\sigma_1(IND) = ind$ and $Mth(pp) = \texttt{ret}\ ind$ hold for some ind, $lvsty_1(ind) \neq \bot$ for the component $lvsty_1$ in $prgty_1(pp)$, and at least one of the following is true:

1. $lvsty_1(ind)$ is not of the form $sbr(\cdots)$.
2. $lvsty_1(ind) = lvsty'_1(ind')$ for some pp' and ind' with $Mth(pp') = \texttt{ret}\ ind'$, $pp \neq pp'$ and $prgty_1(pp') \neq \bot$, where $lvsty'_1$ is in $prgty_1(pp')$.

By Lemma 6, there is a variable assignment σ_2 such $\sigma_2 \sqsupseteq \sigma_1$ holds and rule (T-10) is applicable under σ_2. This means that $\sigma_2(P) = pp$, $\sigma_2(\Phi) = prgty_2$, $prgty_2(pp) \neq \bot$, $\sigma_2(IND) = ind$ and $lvsty_2 \sqsupseteq lvsty_1$ for the $lvsty_2$ in $prgty_2(pp)$.

In the case 1 above, by $lvsty_1(ind) \neq \bot$ and Lemma 1(3), $lvsty_2(ind)$ is not of the form $sbr(\cdots)$. Thus rule (T-10) is applicable but not pre-satisfied at pp with respect to $prgty_2$.

Assume that the case 1 is not true, i.e. that $lvsty_1(ind)$ is of the form $sbr(\cdots)$. Then the case 2 must be true, i.e. $lvsty_1(ind) = lvsty'_1(ind')$ holds for some pp' and ind' as required in the case 2, where $lvsty'_1$ is in $prgty_1(pp')$.

From now on we assume that $lvsty_2(ind)$ is of the form $sbr(\cdots)$; otherwise we know that rule (T-10) is not pre-satisfied at pp with respect to $prgty_2$ and thus we are done.

Now consider whether $lvsty_2(ind) = lvsty'_2(ind')$ holds for the same pp' and ind' in the 2 and $lvsty'_2$ in $prgty_2(pp')$. If it holds, then rule (T-10) is applicable but not pre-satisfied at pp with respect to $prgty_2$, and thus we are done. If it does not hold, then since $lvsty_2(ind)$ is of the form $sbr(\cdots)$, $lvsty'_2(ind')$ is not. Since $Mth(pp') = \texttt{ret}\ ind'$, rule (T-10) is applicable but not pre-satisfied at pp' with respect to $prgty_2$. Hence, the assertion of the lemma always holds. □

Let us now state a property of the operation $\sqcap$ on variable assignments.

Lemma 8. *For every typing rule, if the typing rule is applicable under a variable assignment σ and pre-satisfied under two variable assignments σ_i with $\sigma_{i|\mathcal{FV}(\mathcal{AC})\cup\{\Phi\}} \sqsupseteq \sigma$ for $i = 1, 2$, then $\sigma_1 \sqcap \sigma_2$ denotes a variable assignment, and the typing rule is pre-satisfied under $\sigma_1 \sqcap \sigma_2$.*

Proof. The proof follows by examining each typing rule. Let σ' denote $\sigma_1 \sqcap \sigma_2$. By the assumption given in the assertion of the lemma and the definition of $\sigma_1 \sqcap \sigma_2$, we have that $\sigma'_{|\mathcal{FV}(\mathcal{AC})\cup\{\Phi\}} \sqsupseteq \sigma$ and $\sigma_i \sqsupseteq \sigma'$, in particular, $\sigma_i(P) = \sigma'(P) = \sigma(P)$ and $\sigma_i(\Phi) \sqsupseteq \sigma'(\Phi) \sqsupseteq \sigma(\Phi)$ for $i = 1, 2$.

Consider rule (T-10). It is obviously applicable under $\sigma'_{|\mathcal{FV}(\mathcal{AC})\cup\{\Phi\}}$. Since rule (T-10) is pre-satisfied under σ_i and $\sigma_i \sqsupseteq \sigma'$ holds for $i = 1, 2$, by Lemma 7, rule (T-10) is pre-satisfied under σ'.

Since $\sigma'_{|\mathcal{FV}(\mathcal{AC})\cup\{\Phi\}} \sqsupseteq \sigma$, by Lemma 6(2), there is a variable assignment σ'' such that $\sigma''(\Phi) = \sigma'(\Phi)$, $\sigma''(P) = \sigma(P)$ and $\sigma''(P') = \sigma(P')$ hold, and by the proof for rule (T-10), rule (T-11) is applicable and thus pre-satisfied under σ''. By Lemma 4(2), we have that $\sigma'' = \sigma'$.

The proofs for other typing rules are easy. □

For a program type *prgty*, we define that $reach_{prgty}(pp, pp')$ holds if and only if there are a typing rule and a variable assignment σ such that the rule is applicable under σ, $\sigma(\Phi) = prgty$, $\sigma(P) = pp$ and $\sigma(Pp) = pp'$ for some $\Phi(Pp) \sqsupseteq Ptty \in \mathcal{SC}$.

In general, we write $reach^*_{prgty}(pp, pp')$ to denote that $reach_{prgty}(pp_i, pp_{i+1})$ hold for $i = 1, \cdots, n$ with $n \geq 0$ with $pp = pp_1$ and $pp_{n+1} = pp'$. Note that if $n = 0$ then $pp = pp'$.

Intuitively, the relation $reach^*_{prgty}(pp, pp')$ means that there may exist an execution path from the program point pp to the program point pp' with respect to the program type *prgty*. The program type *prgty* is necessary in determining the path, since a `ret` needs a static type of the given local variable to determine the successor program point.

The following lemma states that the reachability defined as above is preserved under the increase of the involved program type unless rule (T-10) is applicable but not pre-satisfied.

Lemma 9. *If $reach_{prgty}(pp, pp')$ holds for two program points pp and pp′ and a program type prgty, then $reach_{prgty'}(pp, pp')$ holds for every program type prgty′ with $prgty' \sqsupseteq prgty$, unless rule (T-10) is applicable but not pre-satisfied at a program point with respect to prgty′.*

Proof. Follows from the definition of $reach_{prgty}$ and $reach'_{prgty}$, and Lemma 6. □

As another property of the reachability, the following lemma states that if the method *Mth* has a program type, then each program point reachable from the beginning of *Mth* with respect to the program type has a program point type that is not $\perp$ (and thus does not contain $\perp$, either).

Lemma 10. *Let prgty be a program type of the method Mth. If $reach^*_{prgty}(-1, pp)$ holds for a program point $pp \geq 0$, then $prgty(pp) \neq \perp$.*

Proof. Follows directly from the definition of $reach^*_{prgty}$ and the forms of the typing rules. □

As will be shown later, one reason to let the algorithm *analyzer* start with the program type $\perp$ is to ensure that it computes the smallest program type of the method *Mth*, if there is any. A nice by-product is that if *analyzer* yields a *prgty*, then a program point *pp* with $prgty(pp) = \bot$ is always one that is not reachable in *Mth*. This means that *analyzer* also computes dead code in *Mth*.

The procedure *find_a_subst* always yields a variable assignment that is the smallest one in some sense, or a failure.

Lemma 11. *Let find_a_subst take as arguments a typing rule and a variable assignment σ such that the rule is applicable under σ. If there is a variable assignment σ' such that $\sigma'_{|\mathcal{FV}(\mathcal{AC})\cup\{\Phi\}} \sqsupseteq \sigma$ holds and the typing rule is pre-satisfied under σ', then the procedure yields the smallest one among all these variable assignments; otherwise, i.e. if there are no variable assignments like σ' in the above, then the procedure yields a failure.*

Proof. Follows directly from the description in Figure 13 and Lemma 8. □

It is quite easy to prove that the sequence of intermediate program types is an ascending chain.

Lemma 12. *If analyzer does not yield a failure, then it produces a finite ascending chain of intermediate program types $prgty_0 \sqsubset prgty_1 \sqsubset \cdots \sqsubset prgty_n$.*

Proof. Assume that $prgty_k$ is the last obtained intermediate program type in the execution of *analyzer*. Choose a typing rule and a variable assignment σ as required in *analyzer*. By Lemma 11, if it does not fail, then *find_a_subst* in *apply_a_rule* yields a variable assignment σ' with $\sigma'(\Phi) \sqsupseteq \sigma(\Phi)$. Furthermore, if it does not fail, *apply_a_rule* yields the program type

$$prgty_{k+1} = \sigma'(\Phi)[\sigma'(Pp) \mapsto \sigma'(\Phi(Pp)) \sqcup \sigma'(Ptty) \mid \Phi(Pp) \sqsupseteq Ptty \in \mathcal{SC}]$$

Since $\sigma'(\Phi(Pp)) \sqcup \sigma'(Ptty) \sqsupseteq \sigma'(\Phi(Pp))$ always holds, $prgty_{k+1} \sqsupseteq \sigma'(\Phi)$ and thus $prgty_{k+1} \sqsupseteq \sigma(\Phi) = prgty_k$. Thus if $prgty_{k+1} \neq prgty_k$, then $prgty_{k+1} \sqsupset prgty_k$.

The chain $prgty_0 \sqsubset prgty_1 \sqsubset \cdots$ is always finite, since there are only finitely many program types. □

For the further discussion, we need the following lemmas.

Lemma 13. *Assume that there are a typing rule and two variable assignments σ and σ' such that the typing rule is applicable under σ, find_a_subst takes the typing rule and σ as arguments and yields σ' as result. Then for all program points pp, if $\sigma'(\Phi)(pp)$ contains $sbr(sb)$ for some sb, then $\sigma(\Phi)(pp)$ contains $sbr(sb)$, too.*

Proof. By Lemma 11, $\sigma'_{|\mathcal{FV}(\mathcal{AC})\cup\{\Phi\}} \sqsupseteq \sigma$, where $\sigma'(P) = \sigma(P)$ and $\sigma'(\Phi) \sqsupseteq \sigma(\Phi)$. Let *pp* be an arbitrary program point, $\sigma'(\Phi)(pp) = (lvsty', stkty', \cdots)$

and $\sigma(\Phi)(pp) = (lvsty, stkty, \cdots)$. Assume that $lvsty'(ind) = sbr(sb)$ for some ind and sb but $lvsty'(ind) \neq lvsty(ind)$. Now by the $\sqsupseteq$ relation on static types, $lvsty(ind) = \perp$ and thus $\sigma(\Phi)(pp) = \perp$. (Note that we could also have assumed that an entry in $stkty'$ is $sbr(sb)$ but the entry at the same place in $stkty$ is. Then $\sigma(\Phi)(pp) = \perp$ should be true, too.)

Now let $prgty''$ be the program type obtained from $\sigma'(\Phi)$ such that $prgty''(pp) = \perp$ and $prgty''(pp') = \sigma'(\Phi)(pp')$ for all pp' with $pp' \neq pp$. Then it is easy to check that if a typing rule and a variable assignment σ can be used, there is always a variable assignment σ'' such that $\sigma''_{|\mathcal{FV}(\mathcal{AC})\cup\{\Phi\}} \sqsupseteq \sigma$ with $\sigma''(P) = \sigma(P)$. Hence $find\_a\_subst$ must yield σ'' as a result. The contradiction implies that the assertion of the lemma holds. (Actually, since $\sigma(\Phi)(pp) = \perp$, we can only use a typing rule with $\sigma(P) \neq pp$ except that the rule is rule 7. In fact, rule 10 is the only typing rule talking about a subroutine type at program points P' that are not P. But since $\Phi(P') = \Pi'[LT']$ in the condition part of the constraint $\forall P'\forall IND'\forall\Pi'\forall LT'.\cdots$ implies that $\Phi(P') \neq \perp$ the rule allows $\sigma(\Phi)(P') = \perp$.) □

A program point type is said to *contain* a subroutine type $sbr(sb)$ if and only if the program point type is of the form $(lvsty, stkty, \cdots)$ and $lvsty$ or $stkty$ contains $sbr(sb)$.

A program point type is said to *record* a subroutine sb if and only if the program point type is of the form $(\cdots, mod)$ and $sb \in \mathcal{D}om(mod)$ holds.

Lemma 14. *For any intermediate program type $prgty_k$ generated by the algorithm analyzer, if there is a program point pp such that $prgty_k(pp)$ contains a subroutine type $sbr(sb)$ for some subroutine sb, then at least one of the following two assertions holds:*

1. *There are program points pp_i for all $i = 1, \cdots, n+1$ with $n \geq 0$, $sb = pp_1$ and $pp = pp_{n+1}$ such that $reach_{prgty_k}(pp_i, pp_{i+1})$ hold for $i = 1, \cdots, n$, and $prgty_k(pp_i)$ record sb for $i = 1, \cdots, n+1$. (Intuitively, this assertion means that there is a path from sb to pp that is completely inside the subroutine sb.)*
2. *Rule (T-10) is applicable but not pre-satisfied at some program point with respect to $prgty_k$.*

Proof. Let pp be a program point and sb a subroutine. We proceed by induction on the number k. First of all, since $prgty_0(pp) = \perp$, $prgty_0(pp)$ does not contain $sbr(sb)$. Thus the assertion of the lemma holds trivially for $prgty_0$. Assume that the assertion holds for $prgty_g$. Now we prove that the assertion holds for $prgty_{g+1}$.

By Lemma 12, $prgty_{g+1} \sqsupset prgty_g$.

In general, if the assertion 2 holds for $prgty_g$, then by Lemma 7, the assertion 2 holds for $prgty_{g+1}$. Thus we need only to consider the induction assumption where the assertion 2 does not hold for $prgty_g$.

Assume that $prgty_{g+1}(pp)$ contains $sbr(sb)$. We distinguish between the cases whether $prgty_g(pp)$ contains $sbr(sb)$ or not. If it does, then by induction assumption, the assertion 1 holds for $prgty_g$, $pp_0, \cdots, pp_{h+1}$. Since $reach_{prgty_g}(pp_i, pp_{i+1})$, by Lemma 9, $reach_{prgty_{g+1}}(pp_i, pp_{i+1})$, or the assertion 2 holds for $prgty_{g+1}$. Remember that for any mod and mod', $mod' \sqsupseteq mod$ implies that $\mathcal{D}om(mod') = \mathcal{D}om(mod)$. Thus by $prgty_{g+1} \sqsupset prgty_g$, if $prgty_g(pp_i)$ records sb then $prgty_{g+1}(pp_i)$ does. Hence the assertion 1 holds for $prgty_{g+1}$, $pp_0, \cdots, pp_{h+1}$, or the assertion 2 holds for $prgty_{g+1}$.

By the way how *analyzer* works, there are a typing rule and two variable assignments σ and σ' such that $\sigma(\Phi) = prgty_g$ holds, the typing rule is applicable under σ, *find_a_subst* takes the typing rule and σ as arguments and yields σ' as result, $\sigma' \sqsupseteq \sigma$ holds, and *apply_a_rule* yields

$$prgty_{g+1} = \sigma'(\Phi)[\sigma'(Pp) \mapsto \sigma'(\Phi(Pp)) \sqcup \sigma'(Ptty) \mid \Phi(Pp) \sqsupseteq Ptty \in \mathcal{SC}]$$

with $\sigma'(\Phi(Pp)) \sqcup \sigma'(Ptty) \neq \top$.

Now assume that $prgty_g(pp)$ does not contain $sbr(sb)$. Then $\sigma'(\Phi)(pp)$ does not contain $sbr(sb)$; otherwise, by Lemma 13, $prgty_g(pp)$ would contain $sbr(sb)$. Since $\sigma'(\Phi)(pp)$ does not contain $sbr(sb)$, there is $\Phi(Pp) \sqsupseteq Ptty \in \mathcal{SC}$ in the typing rule such that $\sigma'(Pp) = pp$; otherwise, $prgty_{g+1}(pp) = \sigma'(\Phi)(pp)$ would hold, and $\sigma'(\Phi)(pp)$ would contain $sbr(sb)$.

Since there is $\Phi(Pp) \sqsupseteq Ptty \in \mathcal{SC}$ in the typing rule such that $\sigma'(Pp) = pp$, $prgty_{g+1}(pp) = \sigma'(\Phi)(pp) \sqcup \sigma'(Ptty)$. Since $prgty_{g+1}(pp)$ contains $sbr(sb)$ but $\sigma'(\Phi)(pp)$ does not, by Lemma 1, $\sigma'(\Phi)(pp) = \bot$ and $prgty_{g+1}(pp) = \sigma'(Ptty)$. Thus $\sigma'(Ptty)$ contains $sbr(sb)$. Now we consider what the typing rule is:

- The typing rule cannot be rule (T-8) nor (T-10), since they have no $Ptty$. It cannot be rule (T-7), either, since the $Ptty$ never contains $sbr(sb)$.
- Assume that the typing rule is any typing rule that is not rule (T-7), (T-8), (T-9), (T-10) or (T-11). Then by the form of the typing rule, since $\sigma'(Ptty)$ contains $sbr(sb)$, $\sigma'(\Phi(P))$ contains $sbr(sb)$. By Lemma 11, $\sigma(\Phi(P))$, i.e. $prgty_g(\sigma(P))$, contains $sbr(sb)$. By induction assumption, the assertion 1 holds for $prgty_g$, $pp_1, \cdots, pp_{h+1}$ with $pp_1 = sb$ and $pp_{h+1} = \sigma(P)$. Let $pp_{h+2} = \sigma(Pp)$. Then $prgty_{g+1}(pp_{h+2}) = \sigma'(Ptty)$. Since the typing rule is applicable under σ with $\sigma(P) = pp_{h+1}$ and $\sigma(Pp) = pp_{h+2}$, $reach_{prgty_g}(pp_{h+1}, pp_{h+2})$. Since $reach_{prgty_g}(pp_i, pp_{i+1})$ for $i = 1, \cdots, h+1$, by $prgty_{g+1} \sqsupset prgty_g$ and Lemma 9, $reach_{prgty_{g+1}}(pp_i, pp_{i+1})$ for $i = 1, \cdots, h+1$ or the assertion 2 holds for $prgty_{g+1}$. Since $prgty_g(pp_i)$ records sb for $i = 1, \cdots, h+1$, by $prgty_{g+1} \sqsupset prgty_g$, $prgty_{g+1}(pp_i)$ records sb for $i = 1, \cdots, h+1$. By the form of $Ptty$, $prgty_{g+1}(pp_{h+2})$ records sb. Hence the assertion 1 holds for $prgty_{g+1}$, $pp_1, \cdots, pp_{h+2}$, with $pp_1 = sb$ and $pp_{h+2} = \sigma(Pp) = pp$, or 2 holds for $prgty_{g+1}$.
- Assume that the typing rule is rule (T-9). We consider whether $\sigma(SB) = sb$ holds for the SB in the rule. If $\sigma(SB) = sb$, then $pp = sb$ holds, thus the assertion 1 holds trivially. If $\sigma(SB) \neq sb$, since $\sigma'(Ptty)$ contains

$sbr(sb)$, $\sigma'(\Phi(P))$ contains $sbr(sb)$. Now the proof in the previous case applies here.

– Assume that the typing rule is rule (T-11). Since $\sigma'(Ptty)$ contains $sbr(sb)$, $\sigma'(\Phi(P))$ or $\sigma'(\Phi(P'))$ for P and P' in the rule contains $sbr(sb)$. By Lemma 13, $prgty_g(\sigma(P))$ or $prgty_g(\sigma(P'))$ contains $sbr(sb)$.
  - Assume that $prgty_g(\sigma(P))$ contains $sbr(sb)$. By induction assumption, the assertion 1 holds for $prgty_g$, $pp_1, \cdots, pp_{h+1}$ with $sb = pp_1$ and $pp_{h+1} = \sigma(P)$. Let $pp_{h+2} = \sigma(Pp)$.
    Since rule (T-11) is applicable under σ, $reach_{prgty_g}(\sigma(P), \sigma(Pp))$, i.e. $reach_{prgty_g}(pp_{h+1}, pp_{h+2})$. Thus by $prgty_{g+1} \sqsupset prgty_g$ and Lemma 9, $reach_{prgty_{g+1}}(pp_i, pp_{i+1})$ hold for $i = 1, \cdots, h+1$, or the assertion 2 holds for $prgty_{g+1}$.
    Again by $prgty_{g+1} \sqsupset prgty_g$, since the assertion 1 holds as above, $prgty_{g+1}(pp_i)$ records sb for $i = 1, \cdots, h+1$. Since $prgty_{g+1}(\sigma(Pp)) = \sigma'(Ptty)$ and $\sigma'(Ptty)$ contains $sbr(sb)$, by the form of the $Ptty$, $sb \notin sbsin(\sigma'(M), \sigma'(SB))$ and thus $sb \in \mathcal{D}om(modtill(\sigma'(M), \sigma'(SB)))$ hold, where $prgty_{g+1}(pp_{h+1})$ is of the form $(\cdots, \sigma'(M))$. Therefore, $prgty_{g+1}(pp_{h+2})$ records sb.
    Hence, the assertion 1 holds for $prgty_{g+1}$, $pp_1, \cdots, pp_{h+2}$, with $sb = pp_1$ and $pp_{h+2} = \sigma(Pp) = pp$, or the assertion 2 holds for $prgty_{g+1}$.
  - Assume that $prgty_g(\sigma(P'))$ contains $sbr(sb)$. Let pp' stand for $\sigma(P')$ and sb' for $\sigma(SB)$ for the P' and SB in rule (T-11). By induction assumption, the assertion 1 holds for $prgty_g$ and $pp_1, \cdots, pp_{h+1}$ with $sb = pp_1$ and $pp_{h+1} = pp'$.
    Since rule (T-11) is applicable under σ, by the form of the $\mathcal{AC}$ in rule (T-11), $prgty_g(\sigma(P))$ contains $sbr(sb')$, and $Mth(pp') = \texttt{jsr}\ sb'$ holds. Since $prgty_g(\sigma(P))$ contains $sbr(sb')$, by induction assumption, the assertion 1 holds for $prgty_g$, $pp'_1, \cdots, pp'_{f+1}$ and with $pp'_1 = sb'$ and $pp'_{f+1} = \sigma(P)$. Since $Mth(pp') = \texttt{jsr}\ sb'$, rule (T-9) is applicable at pp' with respect to $prgty_g$, and thus $prgty_g(pp', sb')$. Let pp''_i for $i = 1, \cdots, h+f+2$ such that $pp''_i = pp_i$ for $i = 1, \cdots, h+1$ and $pp''_{h+1+i} = pp'_i$ for $i = 1, \cdots, f+1$. Thus $reach_{prgty_g}(pp''_i, pp''_{i+1})$ for $i = 1, \cdots, h+f+1$ with $sb = pp''_1$ and $pp''_{h+f+2} = \sigma(P)$.
    Since $Mth(pp') = \texttt{jsr}\ sb'$, assuming that $prgty_g(pp') = (\cdots, mod)$ and $prgty_g(sb') = (\cdots, mod')$, by the form of rule (T-9), $mod' = mod + (sb', \{\})$. Since $prgty_g(pp')$ records sb, $sb \notin sbsin(mod', sb')$ and $sb \in \mathcal{D}om(modtill(mod', sb'))$. Since $prgty_g(pp'_i)$ record sb' for all $i = 1, \cdots, f+1$, $prgty_g(pp'_i)$ record sb for all $i = 1, \cdots, f+1$.
    By $prgty_{g+1} \sqsupset prgty_g$, the assertion 1 holds for $prgty_{g+1}$, $pp''_1, \cdots, pp''_{h+f+2}$ with $pp''_1 = sb$ and $pp''_{h+f+2} = \sigma(P) = \sigma(Pp) = pp$, or the assertion 2 holds for $prgty_{g+1}$,

□

Roughly speaking, rule (T-11) considers three program points P, SB and P'. The boolean function *monoton_assure* imposes a relation between (the static types at) the program points P and P'. We break down the relation into two simpler relations: one between P and SB, and the other between SB and P'. We model the relation between P and SB using the boolean function *prim_monoton_assure* defined in Figure 16. The boolean function *prim_monoton_assure* is simpler than the boolean function *monoton_assure*, since *prim_monoton_assure* considers program points within one subroutine and thus can be defined independently of the variable assignment used in the application of rule (T-11).

Input: A program type *prgty*, a program point *pp* and a subroutine *sb*.
Output: *true* or *false*.
Body: Assume that $prgty(pp) = (lvsty, \cdots, mod)$ and $prgty(sb) = (lvsty'', \cdots)$. If $sb \in \mathcal{D}om(mod)$ and

$$\forall 0 \leq j < LocN. j \notin mlvsin(sb, mod) \Rightarrow lvsty(j) \sqsupseteq lvsty''(j)$$

hold, then the current function yields *true*; otherwise it yields *false*.

Figure 16: The boolean function *prim_monoton_assure*

The next lemma states a condition that ensures that the boolean function *prim_monoton_assure* yields *true*.

Lemma 15. *Assume that prgty is a fixed point and that $pp_1, \cdots, pp_{n+1}$ are program points such that $reach_{prgty}(pp_i, pp_{i+1})$ hold for $i = 1, \cdots, n$, $prgty(pp_i)$ records pp_1 for $i = 1, \cdots, n+1$, and $mod_1 = [\cdots, (pp_1, \{\})]$ for $prgty(pp_1) = (\cdots, mod_1)$. Then $prim\_monoton\_assure(prgty, pp_{n+1}, pp_1)$ yields true.*

Proof. Let $prgty(pp_i) = (lvsty_i, \cdots, mod_i)$ for $i = 1, \cdots, n+1$. Let sb denote pp_1. Since $prgty(pp_i)$ record sb for $i = 1, \cdots, n+1$, $sb \in \mathcal{D}om(mod_i)$ hold and $mlvsin(mod_i, sb)$ are defined for all $i = 1, \cdots, n+1$.

Since *prgty* is a fixed point, by the forms of all those typing rules that may induce $reach_{prgty}(pp_i, pp_{i+1})$, $mlvsin(mod_{i+1}, sb) \supseteq mlvsin(mod_i, sb)$ hold for all $i = 1, \cdots, n$. In particular, if the typing rule is rule (T-11), then the term $addmlvs(mlvsin(M, SB), modtill(M, SB))$ in the rule assures the assertion.

Now we prove that

$$\forall 0 \leq j < LocN. j \notin mlvsin(mod_i, sb) \Rightarrow lvsty_i(j) \sqsupseteq lvsty_1(j)$$

hold for all $i = 1, \cdots, n+1$. We proceed by induction on n.

If $n = 0$, then the assertion holds trivially. Assume that the assertion holds for $n = m - 1$ with $m \geq 1$. Now let $n = m$. Let j be an arbitrary number with $0 \leq j < LocN$ and $j \notin mlvsin(sb, mod_{m+1})$. Since $mlvsin(mod_{i+1}, sb) \supseteq mlvsin(mod_i, sb)$ for $i = 1, \cdots, m$, $j \notin mlvsin(mod_i, sb)$ hold for all $i = 1, \cdots, m+1$. Now consider all those typing rules that may induce $reach_{prgty}(pp_m, pp_{m+1})$.

If the typing rule is not rule (T-11), then it is straightforward to see that $lvsty_{m+1}(j) \sqsupseteq lvsty_m(j)$ holds. By induction assumption, $lvsty_m(j) \sqsupseteq lvsty_1(j)$. Thus $lvsty_{m+1}(j) \sqsupseteq lvsty_1(j)$.

Assume the typing rule is rule (T-11). Since *prgty* is a fixed point, there is a variable assignment σ with $\sigma(\Phi) = prgty$, $\sigma(P) = pp_m$, $\sigma(P') + 1 = pp_{m+1}$ and $Mth(\sigma(P')) = \texttt{jsr}\ \sigma(SB)$ such that rule (T-11) is satisfied under σ. Since $mod_1 = [\cdots, (pp_1, \{\})]$ and $prgty(pp_i)$ record sb for $i = 1, \cdots, m+1$, $sb \in \mathcal{D}om(modtill(mod_m, \sigma(SB)))$, and thus $\sigma(SB) \in sbsin(mod_m, sb)$. Therefore, there is k with $1 \leq k \leq m$ such that $pp_k = \sigma(P')$ and $pp_{k+1} = \sigma(SB)$. Since rule (T-11) is satisfied under σ,

$$lvsty_{m+1} \sqsupseteq \\ lvsty_k[i \mapsto turnUnus(lvsty_m(i), sbsin(mod_m, \sigma(SB)) \mid i \in mlvsin(mod_m, \sigma(SB))]$$

holds. Since $j \notin mlvsin(mod_{m+1}, sb)$ holds, $lvsty_{m+1}(j) \sqsupseteq lvsty_k(j)$. By induction assumption, $lvsty_k(j) \sqsupseteq lvsty_1(j)$. Thus $lvsty_{m+1}(j) \sqsupseteq lvsty_1(j)$.
□

Now we formally state that the restriction of the application of rule (T-11) in the algorithm *analyzer* is not a serious restriction.

Lemma 16. *Assume that analyzer terminates with a program type prgty. If there is a variable assignment σ with $\sigma(\Phi) = prgty$ such that rule (T-11) is applicable under σ, then monoton_assure(σ) yields true.*

Proof. Since *analyzer* terminates with *prgty*, all typing rules that are applicable are pre-satisfied.

Given the notation in rule (T-11), since rule (T-11) is applicable under σ, $prgty(\sigma(P)) = (\sigma(LT), \cdots)$, $\sigma(LT(IND)) = sbr(\sigma(SB))$ and $Mth(\sigma(P')) = \texttt{jsr}\ \sigma(SB)$ hold. By Lemma 14, there exist pp_i for all $i = 1, \cdots, n+1$ with $n \geq 0$, $\sigma(SB) = pp_1$ and $\sigma(P) = pp_{n+1}$ such that $reach_{prgty_k}(pp_i, pp_{i+1})$ hold for $i = 1, \cdots, n$, and $prgty_k(pp_i)$ record sb for $i = 1, \cdots, n+1$.

Let pp_0 denote $\sigma(P')$. Let $prgty(pp_i)$ be of the form $(lvsty_i, \cdots, mod_i)$ for $i = 0, \cdots, n+1$. Since $Mth(pp_0) = \texttt{jsr}\ pp_1$, $mod_1 = [\cdots, (pp_1, \{\})]$. By Lemma 15, *prim_monoton_assure*$(prgty, pp_{n+1}, pp_1)$ yields *true*.

Since $Mth(pp_0) = \texttt{jsr}\ pp_1$, by the form of rule (T-9), $lvsty_1 \sqsupseteq lvsty_0$. Therefore

$$\forall 0 \leq j < LocN. j \notin mlvsin(pp_1, mod_{n+1}) \Rightarrow lvsty_{n+1}(j) \sqsupseteq lvsty_0(j)$$

holds. Hence, *monoton_assure*(σ) yields *true*. □

8.2 Correctness of the algorithm *analyzer*

Now we can prove that *analyzer* yields a fixed point.

Lemma 17. *If analyzer terminates with a program type prgty, then prgty is a fixed point.*

Proof. We need to prove that whenever a typing rule is applicable under a variable assignment σ and $\sigma(\Phi) = prgty$ holds, there is a variable assignment σ' such that the typing rule is satisfied under σ' and $\sigma'_{|\mathcal{FV}(\mathcal{AC})\cup\{\Phi\}} = \sigma$ holds.

Assume that there is a typing rule and a variable assignment σ as required. By Lemma 16, either the typing rule is not rule (T-11), or $monoton\_assure(\sigma)$ yields *true*. Thus *analyzer* must have called *apply_a_rule* and *find_a_subst*, *find_a_subst* must have computed a variable assignment σ', and *apply_a_rule* must have yielded

$$\sigma'(\Phi)[\sigma'(Pp)\mapsto\sigma'(\Phi(Pp))\sqcup\sigma'(Ptty) \mid \Phi(Pp)\sqsupseteq Ptty\in\mathcal{SC}]$$

which is equal to *prgty* and saitisfies that $\sigma'(\Phi(Pp))\sqcup\sigma'(Ptty) \neq \top$. By Lemma 11, $\sigma'_{|\mathcal{FV}(\mathcal{AC})\cup\{\Phi\}} \sqsupseteq \sigma$ and the typing rule is pre-satisfied under σ'. By the standard property of $\sqcup$, $\sigma'(\Phi(Pp)) \sqsupseteq \sigma'(Ptty)$ and $\sigma'(\Phi) = prgty = \sigma(\Phi)$ hold. The fact that $\sigma'(\Phi(Pp)) \sqsupseteq \sigma'(Ptty)$ hold for all $\Phi(Pp)\sqsupseteq Ptty \in \mathcal{SC}$ implies that the typing rule is satisfied under σ'. By Lemma 5, $\sigma'(\Phi) = \sigma(\Phi)$ implies that $\sigma'_{|\mathcal{FV}(\mathcal{AC})\cup\{\Phi\}} = \sigma$. Thus the assertion of the lemma holds. □

The correctness of *analyzer* is formulated in the following theorem.

Theorem 18. *(Correctness) If analyzer yields a program type prgty, then the method Mth has the program type prgty.*

Proof. Let us use the notations in the definition of program types of *Mth* (in Section 5). Consider an arbitrary typing rule and a variable assignment σ with $\mathcal{D}om(\sigma) = Q\cup\{\Phi\}$ and $\sigma(\Phi) = prgty$ such that $\mathcal{AC}$ is satisfied under σ. Thus the typing rule is applicable under σ. By Lemma 17, *prgty* is a fixed point. By the definition of fixed points, there is a variable assignment σ' such that the typing rule is satisfied under σ' and $\sigma'_{|\mathcal{FV}(\mathcal{AC})\cup\{\Phi\}} = \sigma$ holds. Therefore, $\mathcal{D}om(\sigma') = Q\cup Q'\cup\{\Phi\}$ holds and $\mathcal{CC}\cup\mathcal{SC}$ is satisfied under σ'. Since the typing rule and σ have been arbitrarily chosen, the assertion of the theorem holds. □

8.3 Termination of the algorithm *analyzer*

The termination of the algorithm *analyzer* is easy to prove.

First of all, the termination of *find_a_subst* and *apply_a_rule* can be easily established.

Lemma 19. *If all defined functions used in typing rules are terminating, then find_a_subst is terminating.*

Proof. Follows from the facts that there are only finitely many variable assignments and that the check of constraints is terminating. □

Lemma 20. *If all defined functions used in typing rules are terminating, then apply_a_rule is terminating.*

Proof. Follows from the fact that the check of constraints is terminating and from Lemma 19. □

Theorem 21. *(Termination) If all defined functions used in typing rules are terminating, then analyzer is terminating.*

Proof. Follows easily from Lemmas 20 and 12. □

8.4 Completeness of the algorithm *analyzer*

The first thing is to prove that each program type of *Mth* is a fixed point.

Lemma 22. *If the method Mth has a program type, then the program type is a fixed point.*

Proof. Let us use the notations at the end of Section 5. Assume that the method *Mth* has a program type *prgty*. Consider an arbitrary typing rule and an arbitrary σ with $\sigma(\Phi) = prgty$ such that the typing rule is applicable under σ. Then $\mathcal{D}om(\sigma) = Q \cup \{\Phi\}$ holds and $\mathcal{AC}$ is satisfied under σ. Since the method *Mth* has the program type *prgty*, there is a variable assignment σ' such that $\mathcal{D}om(\sigma') = Q \cup Q' \cup \{\Phi\}$ and $\sigma'_{|Q \cup \{\Phi\}} = \sigma$ hold and $\mathcal{CC} \cup \mathcal{SC}$ is satisfied under σ'. Thus the typing rule is satisfied under σ'. Hence, *prgty* is a fixed point restricted to the typing rule and σ. Since the typing rule and σ have been arbitrarily chosen, the assertion of the lemma holds. □

Now we prove that if the method *Mth* has a program type *prgty*, then *analyzer* will yield a program type *prgty'* with $prgty \sqsupseteq prgty'$. In order to do this, we first prove that *apply_a_rule* is monotone in a certain sense.

The procedure *apply_a_rule* is said to be *monotone with respect to a typing rule and a set of variable assignments* if and only if, for all variable assignments σ_1 and σ_2 from the set such that the typing rule is applicable under σ_1 and σ_2, *apply_a_rule* yields $prgty'_i$ for the typing rule and σ_i for both $i = 1, 2$, if $\sigma_1 \sqsupseteq \sigma_2$, then $prgty'_1 \sqsupseteq prgty'_2$.

Lemma 23. *If it does not terminate with failure, apply_a_rule is monotone with respect to each typing rule that is not rule (T-11) and the set of all possible variable assignments.*

Proof. Consider an arbitrary typing rule that is not rule (T-11). Assume that σ_1 and σ_2 are two variable assignments with $\sigma_1 \sqsupseteq \sigma_2$ such that the typing rule is applicable under σ_1 and σ_2.

First, *apply_a_rule* calls *find_a_subst*. Let *find_a_subst* take the typing rule and σ_1 (or σ_2). If it does not fail, then by Lemma 11, *find_a_subst* yields the smallest one σ_1' (or σ_2', respectively) among all variable assignments σ_1'' (or σ_2'', respectively) such that the typing rule is pre-satisfied under σ_1'' (or σ_2'', respectively), and

$$\sigma''_{1\,|\mathcal{FV}(\mathcal{AC})\cup\{\Phi\}} \sqsupseteq \sigma_1 \text{ (or } \sigma''_{2\,|\mathcal{FV}(\mathcal{AC})\cup\{\Phi\}} \sqsupseteq \sigma_2\text{, respectively)}$$

holds. Since the set of all σ_1'' is a subset of all σ_2'', $\sigma_1' \sqsupseteq \sigma_2'$ holds.

Second, by examining each typing rule, it is easy to check that if $\Phi(Pp) \sqsupseteq Ptty \in \mathcal{SC}$, then $\mathcal{FV}(Pp)$ does not contain variables whose sorts are based on static types. By the definition of $\sigma_1' \sqsupseteq \sigma_2'$, we know that $\sigma_1'(X) = \sigma_2'(X)$ for all $X \in \mathcal{FV}(Pp)$. Thus $\sigma_1(Pp) = \sigma_2(Pp)$.

Third, by examining each typing rule, it is easy to check that $\sigma_1'(Ptty) \sqsupseteq \sigma_2'(Ptty)$ holds for each $\Phi(Pp) \sqsupseteq Ptty \in \mathcal{SC}$.

Hence, we finally have that

$$\begin{array}{l} \sigma_1'(\Phi)[\sigma_1'(Pp) \mapsto \sigma_1'(\Phi(Pp)) \sqcup \sigma_1'(Ptty) \mid \Phi(Pp) \sqsupseteq Ptty \in \mathcal{SC}] \\ \sqsupseteq\ \ \sigma_2'(\Phi)[\sigma_2'(Pp) \mapsto \sigma_2'(\Phi(Pp)) \sqcup \sigma_2'(Ptty) \mid \Phi(Pp) \sqsupseteq Ptty \in \mathcal{SC}] \end{array}$$

□

Lemma 24. *If it does not terminate with failure, apply_a_rule is monotone with respect to rule (T-11) and the set of all those variable assignments σ such that monoton_assure(σ) yields true.*

Proof. Assume that σ_i for $i = 1, 2$ are variable assignments with $\sigma_1 \sqsupseteq \sigma_2$ such that rule (T-11) is applicable under σ_i and *monoton_assure*(σ_i) yield *true* for $i = 1, 2$.

Assume that the procedure *find_a_subst* yields σ_i' for $i = 1, 2$ as described in the proof of Lemma 23. Furthermore, the proof of Lemma 23 applies here, except the proof of $\sigma_1'(Ptty) \sqsupseteq \sigma_2'(Ptty)$. To prove that $\sigma_1'(Ptty) \sqsupseteq \sigma_2'(Ptty)$, let us use the notations in rule (T-11) and the notations satisfying that

$$\begin{array}{ll} sb = \sigma_i'(SB) & \\ \sigma_i'(LT) = lvsty_i, & \sigma_i'(LT') = lvsty_i', \\ \sigma_i'(ST) = stkty_i, & \sigma_i'(ST') = stkty_i', \\ \sigma_i'(M) = mod_i, & \sigma_i'(M') = mod_i', \\ \sigma_i(\Phi(P)) = (lvsty_i, stkty_i, mod_i) & \sigma_i(\Phi(P')) = (lvsty_i', stkty_i', mod_i') \end{array}$$

for $i = 1, 2$. Since $\sigma_1'(\Phi(P)) \sqsupseteq \sigma_2'(\Phi(P))$ and $\sigma_1'(\Phi(P')) \sqsupseteq \sigma_2'(\Phi(P'))$, we have

$$\begin{array}{l} lvsty_1 \sqsupseteq lvsty_2,\ stkty_1 \sqsupseteq stkty_2,\ mod_1 \sqsupseteq mod_2, \\ lvsty_1' \sqsupseteq lvsty_2',\ stkty_1' \sqsupseteq stkty_2',\ mod_1' \sqsupseteq mod_2'. \end{array}$$

Let us define t_i, s_i and m_i for $i = 1, 2$ as follows:

$$t_i = lvsty'_i[j \mapsto turnUnus(lvsty_i(j), sbsin(mod_i, sb)) \mid j \in mlvsin(mod_i, sb)]$$
$$s_i = turnUnus(stkty_i, sbsin(mod_i, sb))$$
$$m_i = addmlvs(mlvsin(mod_i, sb), modtill(mod_i, sb))$$

Then the key is to prove that $t_1 \sqsupseteq t_2$, $s_1 \sqsupseteq s_2$ and $m_1 \sqsupseteq m_2$.

Since $mod_1 \sqsupseteq mod_2$, we have that $mlvsin(mod_1, sb) \supseteq mlvsin(mod_2, sb)$, $sbsin(mod_1, sb) = sbsin(mod_2, sb)$ and $modtill(mod_1, sb) \sqsupseteq modtill(mod_2, sb)$.

In order to prove that $t_1 \sqsupseteq t_2$, we prove that $t_1(j) \sqsupseteq t_2(j)$ for each j. We need only to distinguish between three cases for j:

- If $j \notin mlvsin(mod_i, sb)$ for $i = 1, 2$, then $t_i(j) = lvsty'_i(j)$. By $lvsty'_1 \sqsupseteq lvsty'_2$, $t_1(j) \sqsupseteq t_2(j)$.
- If $j \in mlvsin(mod_1, sb)$ but $j \notin mlvsin(mod_2, sb)$, then

  $$t_1(j) = turnUnus(lvsty_1(j), sbsin(mod_1, sb)) \text{ and}$$
  $$t_2(j) = lvsty'_2(j).$$

  By Lemma 2, $t_1(j) \sqsupseteq lvsty_1(j)$ holds. Since $lvsty_1 \sqsupseteq lvsty_2$, $lvsty_1(j) \sqsupseteq lvsty_2(j)$. Since $monoton\_assure(\sigma_2)$ yields $true$, $lvsty_2(j) \sqsupseteq lvsty'_2(j)$. Hence $t_1(j) \sqsupseteq lvsty'_2(j) = t_2(j)$.
- If $j \in mlvsin(mod_i, sb)$ for $i = 1, 2$, then

  $$t_i(j) = turnUnus(lvsty_i(j), sbsin(mod_i, sb)).$$

  We need only to consider the following cases:
  - Assume that $lvsty_1(j) = sbr(sb')$. By $lvsty_1(j) \sqsupseteq lvsty_2(j) \neq \bot$, $lvsty_2(j) = sbr(sb')$. Since $sbsin(mod_1, sb) = sbsin(mod_2, sb)$, $t_1(j) = t_2(j)$.
  - Assume that $lvsty_1(j) \neq sbr(sb')$. If $lvsty_2(j) = sbr(sb')$, then by $lvsty_1(j) \sqsupseteq lvsty_2(j)$, $lvsty_1(j) = unus$. Hence $t_1(j) \sqsupseteq t_2(j)$. If $lvsty_2(j) \neq sbr(sb')$, then $t_1(j) = lvsty_1(j) \sqsupseteq lvsty_2(j) = t_2(j)$ hold.

The proof for $s_1 \sqsupseteq s_2$ is similar to and simpler than that for $t_1 \sqsupseteq t_2$.

The proof for $m_1 \sqsupseteq m_2$ is straightforward. $\square$

Now we prove that if there is a fixed point, then all intermediate program types produced by *analyzer* are smaller than the fixed point.

Lemma 25. *Assume that $prgty_k$ for $k = 0, 1, \cdots$, be the sequence of intermediate program types produced by analyzer. Let $prgty$ be a fixed point. Then $prgty \sqsupseteq prgty_k$ holds for all $k = 0, 1, \cdots$.*

Proof. We proceed by induction on k.

If $k = 0$, then $prgty \sqsupseteq prgty_0 = \bot$. As the induction assumption we assume that $prgty \sqsupseteq prgty_k$.

Assume that *analyzer* produces an intermediate program type $prgty_{k+1}$. This means that there are a typing rule and a variable assignment σ with $\sigma(\Phi) = prgty_k$ such that the rule is applicable under σ, if the rule is rule (T-11) then the $monoton\_assure(\sigma)$ yields *true*, and *apply_a_rule* yields $prgty_{k+1}$. By Lemma 6, there is a variable assignment σ' such that $\sigma'(\Phi) = prgty$ and $\sigma' \sqsupseteq \sigma$ hold and the typing rule is applicable under σ', unless rule (T-10) is applicable but not pre-satisfied at a program point with respect to *prgty*. Since *prgty* is a fixed point, the "unless"-case is impossible. Thus *apply_a_rule* yields *prgty* for the typing rule and σ'. Since $\sigma' \sqsupseteq \sigma$, by Lemmas 23 and 24, $prgty \sqsupseteq prgty_{k+1}$. □

In order to show that if it is possible for *analyzer* to yield a failure then the method *Mth* has no program types, we need the following lemma:

Lemma 26. *For any typing rule and any variable assignment σ_1, if the typing rule is applicable under σ_1 and apply_a_rule yields a failure for the typing rule and σ_1, then for any program type $prgty'$ with $prgty' \sqsupseteq \sigma_1(\Phi)$, there is a variable assignment σ_2 with $\sigma_2(\Phi) = prgty'$ such that apply_a_rule yields a failure for the same typing rule or rule (T-10) and σ_2.*

Proof. Assume that there are a typing rule, σ_1 and $prgty'$ as required. By Lemma 6, either

1. there is σ_2 with $\sigma_2(\Phi) = prgty'$ and $\sigma_2 \sqsupseteq \sigma_1$ such that the typing rule is applicable under σ_2, or
2. rule (T-10) is applicable but not pre-satisfied at a program point with respect to $prgty'$.

The case 2 directly implies the assertion of the lemma.

Consider the case 1. In general, there are only two possibilities where *apply_a_rule* yields a failure for the typing rule and σ_1. The first is when *find_a_subst* yields a failure for the typing rule and σ_1, i.e. when there are no σ'_1 such that $\sigma'_{1|\mathcal{FV}(\mathcal{AC})\cup\{\Phi\}} \sqsupseteq \sigma_1$ holds and the typing rule is pre-satisfied under σ'_1. Since $\sigma_2 \sqsupseteq \sigma_1$, there are no σ'_2 such that $\sigma'_{2|\mathcal{FV}(\mathcal{AC})\cup\{\Phi\}} \sqsupseteq \sigma_2$ holds and the typing rule is pre-satisfied under σ'_2. Hence *find_a_subst* yields a failure for the typing rule and σ_2.

The second possibility is that *find_a_subst* yields σ'_1 for the typing rule and σ_1, but $\sigma'_1(\Phi(Pp)) \sqcup \sigma'_1(Ptty) = \top$ for some $\Phi(Pp) \sqsupseteq Ptty \in \mathcal{SC}$. If *find_a_subst* yields a variable assignment σ'_2 for the typing rule and σ_2, then since $\sigma'_2 \sqsupseteq \sigma'_1$, $\sigma'_2(\Phi(Pp)) \sqcup \sigma'_2(Ptty) = \top$. Hence *apply_a_rule* yields a failure for the typing rule and σ_2. □

We are ready to prove the completeness, i.e that if *Mth* is statically well-typed, then *analyzer* always yields a program type of *Mth*. Furthermore, the yielded program type is the smallest one among all program types of *Mth*.

Theorem 27. *(Completeness) If Mth has a program type prgty, then analyzer will yield a program type prgty′ of Mth with prgty* $\sqsupseteq$ *prgty′.*

Proof. Assume that *Mth* has a program type *prgty*. By Lemma 22, *prgty* is a fixed point. By Lemma 25, all intermediate program types $prgty_k$ for $k = 0, 1, \cdots$, produced by *analyzer* satisfy that $prgty \sqsupseteq prgty_k$. Since *prgty* is a fixed point, by Lemma 26, *analyzer* does not yield a failure. By Theorems 21, the algorithm will yield a program type $prgty_n$ for some finite n. Note that $prgty \sqsupseteq prgty_n$. By Theorem 18, $prgty_n$ is a program type of *Mth*. □

Note that Theorem 27 implies that if *Mth* has a program type, then *analyzer* will yield the smallest one.

9 Related work

Bertelsen formalized JVM instructions using state transitions [2]. Cohen described a formal semantics of a subset of the JVM, but runtime checks are used to assure type-safe execution [3]. Both approaches did not consider static type check, thus did not directly correspond to bytecode verification.

Stata and Abadi [17] proposed a type system for subroutines, provided lengthy proofs for the soundness of the system and clarified several key semantic issues about subroutines. Freund and Mitchell [5] made a significant extension of Stata and Abadi's type system by considering object initialization. Hagiya and Tozawa [8] presented another type system for subroutines, where the soundness proof is extremely simple. Qian [13] presented a constraint-based typing system for objects, primitive values, methods and subroutines and proved the soundness. Pusch [12] formalized a subset of JVM in the theorem prover Isabelle/HOL and reached a higher degree of reliability. All this work basically aimed at achieving a sound specification, which defines what types memory locations should have, but did not consider how to develop a provably correct implementation to compute canonical types for memory locations. Note that Hagiya and Tozawa discussed issues on an implementation of their type system, but did not formally describe the implementation. In fact, since they did not consider objects, their implementation did not encounter the problems we have here.

Goldberg [7] directly used dataflow analysis to formally specify bytecode verification focusing on type-correctness and global type consistency for dynamic class loading. He successfully formalized a way to relate bytecode verification and class loading. Since he did not consider subroutines, he did not encounter the problems we discuss in this paper.

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

The Kimera project [15] was quite effective in detecting flaws in commercial bytecode verifiers. Following the comparative testing approach, they wrote a reference bytecode verifier and tested commercial bytecode verifiers against it. Since it was built by a simple organization of implementations of individual axioms distilled from the OJVMS, their reference bytecode verifier could be easier understood, revised and debugged, and thus reached a higher degree of correctness than the commercial ones. However, since the axioms are written in English, they might still be hard to reason about, and the correctness of the reference bytecode verifier could not be formally established.

10 Conclusion

Practically, we have shown an approach in which a core of a bytecode verifier can be built. We see no fundamental difficulties to apply the approach to build a bytecode verifier for the entire JVM. Since the correctness, termination and completeness of the bytecode verifier can be formally proved, we could reach a higher degree of reliability. Since the fixed part of the bytecode verifier are basically the algorithm and procedures in Section 7, which are parameterized by the description of (the typing rules of) a formal specification, many changes in the formal specification may cause no or only small changes in the fixed part of the bytecode verifier or the formal proofs, and thus automatically lead to bytecode verifiers for revisions of the formal specification. Therefore, our bytecode verifier could be used as a prototype to test the behaviors of a formal specification.

Theoretically, we have shown results on the existence and effective computation of the smallest program type for JVM programs, corresponding to the results on the existence and the effective computation of principal types for `ML` programs.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.
2. P. Bertelsen. Semantics of java byte code. `http://www.dina.kvl.dk/~pmb/`, 1997.
3. R. Cohen. The Defensive Java Virtual Machine specification. Technical report, Computational Logic inc., 1997.
4. P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs bu construction of approximation of fixpoints. In *Proc. 4th ACM Symp. Principles of Programming Languages*, pages 238–258, 1977.
5. S. Freund and J. Mitchell. A type system for object initialization in the java bytecode language (summary). *Electronic Notes in Theoretical Computer Science*, 10, 1998. `http://www.elsevier.nl/locate/entcs/volume10.html`.
6. J. Gallier. *Logic for Computer Science - Foundations of Automatic Theorem Proving*. John Wiley & Sons, 1987.
7. A. Goldberg. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conference on Computer and Communications Security*, 1998. To appear.

8. M. Hagiya and A. Tozawa. On a new method fot dataflow analysis of Java Virtual Machine subroutines. In *Proc. 1998 Static Analysis Symposium*. Springer-Verlag LNCS, 1998. To appear.
9. J. Jaffar and M. Maher. Constraint logic programming: A survey. *J. of Logic Programming*, pages 503–581, 1994.
10. G. McGraw and E. Felten. *Java Security*. Wiley Computer Publishing, 1997.
11. Neil D.Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program analysis. In T. M. S.Ãbramsky, Dov M. Gabbay, editor, *Handbook of Logic in Computer Science, vol. 4, Semantic Modelling*. Oxford University Press, 1995.
12. C. Pusch. Formalizing the Java Virtual Machine in Isabelle/HOL. Technical report, TUM I9816, Technische Unversität München, 1998. `http://www4.informatik.tu-muenchen.de/~isabelle/bali/`.
13. Z. Qian. A formal specification of Java^{tm} virtual machine instructions for objects, methods and subroutines. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java^{TM}*. Springer Verlag LNCS, 1998. To appear.
14. V. Saraswat. Java is not type-safe. Technical report, AT&T Research, 1997. `http://www.research.att.com/~vj/bug.html`.
15. E. Sirer, S. McDirmid, and B. Bershad. A Java system security architecture. `http://kimera.cs.washington.edu/`, 1997.
16. G. Smolka, W. Nutt, J. Goguen, and J. Meseguer. Order-sorted equational computation. In H. Aït-Kaci and M. Nivat, editors, *Resolution of Equations in Algebraic Structures, volume 2*, pages 297–367. Academic Press, 1989.
17. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proc. 25st ACM Symp. Principles of Programming Languages*, 1998.