Least Types for Memory Locations in Java™ Bytecode (Extended Summary)

Zhenyu Qian

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

Abstract. (Java) bytecode verification should ensure type-safety for Java Virtual Machine (JVM) programs (bytecode) and form a basis for Java-based internet security. Recent research work has proposed using formal typing rules to enforce constraints on types for memory locations in bytecode and thus formally specify (part of) bytecode verification. In this paper, we take one step forward and show that typing rules can be non-deterministically and repeatedly used in an arbitrary order to compute the least types for memory locations satisfying the constraints, if there are any, or to report a failure, if there are none.

The result implies that theoretically a bytecode verifier can have any strategy visiting program points in bytecode without affecting the acceptance of bytecode, i.e. without causing denial of services provided by bytecode. It also suggests ways to implement bytecode verification directly using formal specification.

1 Introduction

Java programs are usually compiled into Java bytecode, which is then interpreted by the Java Virtual Machine (JVM). Java bytecode is a program of *JVM instructions* (short *instructions*) that operate on the *operand stack* (short: the *stack*) and registers, called *local variables*. Stack entries and local variables are called *memory locations* in this paper. Memory locations hold data that are either object references or primitive values. In order for instructions to be executed in a defined way, we need to ensure that data are treated by instructions in a type-safe way. In this paper, we consider arbitrary classes of objects but only one type of primitive values, namely integers.

According to the official JVM specification [5] (hereafter OJVMS), one main task of (Java) bytecode verification is to statically insure type-safe execution of instructions. For this purpose, bytecode verification should introduce certain types that classify all data, and attempt to assign these types to memory locations at program points in bytecode so that the following (*soundness*) results hold:

- Memory locations may never hold data not of the assigned types.
- Instructions accessing memory locations are always defined on the assigned types.

Bytecode is legal and should be accepted by bytecode verification if and only if types can be successfully assigned to all memory locations.

Unfortunately, the OJVMS is ambiguous and does not define in an implementation-independent way when bytecode is legal. Since type-safety forms a basis for Java-based internet security, recent work (e.g. [10,2,8]) has proposed using formal typing rules to formally define legal bytecode and proved the soundness. The work thus specifies (part of) bytecode verification.

In this paper, we show that the typing rules can be non-deterministically and repeatedly used in an arbitrary order to compute the least assigned types if there are any, or to report a failure if there are none. We follow the line in [8], though we strongly conjecture that our result applies to other related work (see Section 8).

The paper is organized as follows: Section 2 intuitively explains what is bytecode and what are types for memory locations. The main problem and outlines of our solution are given in Section 3. Section 4 introduces some basic notations. Section 5 defines the types we will use. The typing rules are presented in Section 6.

Section 7 describes the algorithm and states the main properties. Section 8 discusses related work. Section 9 contains conclusions.

This paper is an extended summary of a full paper that is available under `http://www.kestrel.edu/~qian/abs-jvmdflow.html`.

2 Bytecode

Bytecode comes as so-called *class files*, which are in fact results obtained by compiling Java classes. A class file contains a *method* if and only if the original Java class contains the corresponding Java method.

A *method* contains, among others, a name for the method, types for all parameters and return value, a number of local variables, and a *body*, which is actually a sequence of instructions. The position of an instruction in a body is called a *program point*.

In this paper we consider the following instructions: `aload`, `iload`, `astore`, `istore`, `if_acmpeq`, the implicit instruction dealing with the start of a method, `return`, `jsr` and `ret`. In particular, instructions `jsr` and `ret` build the mechanism for subroutines, typically to implement `finally` clauses in `Java` programs. More concretely, `jsr` *sb* calls a *subroutine sb*, which is in fact a program point, by first pushing the next program point as the return program point onto the stack and then transferring control to *sb*, while `ret` j returns from a subroutine by transferring control to a return program point stored in local variable j.

```
Method void m(int,int)    // Method head.
.limit locals 5   // Allocate space for 5 local variables.
                  // Store this object and the actual parameters into the
                  // first 3 local variables, create an empty stack.
 0 jsr 5          // Push the program point 1 onto the stack and go to 5.
 1 aload 1        // Copy the object in local variable 1 onto the stack
 2 astore 3       // Move the top object from the stack into local variable 3.
 3 jsr 5          // Push the program point 4 onto the stack and go to 5.
 4 return         // The method returns.
 5 astore 4       // Move the top object from the stack into local variable 4.
 6 aload 1        // Copy the object in local variable 1 onto the stack
 7 aload 1        // Copy the object from local variable 1 onto the stack
 8 if_acmpeq 11   // If the two top objects in the stack are equal, go to 11.
 9 aload 1        // Copy the object in local variable 1 onto the stack
10 astore 2       // Move the top object from the stack into local variable 2
11 jsr 12         // Push the program point 12 onto the stack and go to 12.
12 astore 5       // Move the top object from the stack into local variable 5.
13 ret 4          // Return to the program point stored in local variable 4.
```

Figure 1: A JVM program

Consider, for example, the method in Figure 1. In the method, both `jsr 5` at `0` and `3` call the subroutine `5`. The `astore` instruction at `5` moves the return program point `1` or `4` from the stack into local variable `4`. The subroutine `5` calls another subroutine `12` at `11`, but the instruction `ret 3` at `13` brings control from the subroutine `12` directly back to the caller of the subroutine `5`, using the return program point (i.e. `1` or `4`) stored in local variable `4`. In general, a subroutine need not contain a `ret` instruction. If it does, then an `astore` instruction should be used to store the return program point on the stack into a local variable so that `ret` can access it.

2.1 Types for memory locations

As mentioned before, bytecode verification should attempt to assign a kind of types to memory locations in bytecode. Note that these types are not necessary the types in Java or JVM programs.

As an example, Figure 2 gives types assigned to memory locations in the method in Figure 1. Note that the types are not the unique ones, but as we will see later, they are collectively the least ones. We write types for local variables at each program point under LV and those for stack entries under ST. The column MD will be discussed later in this section. The column "Succ's" contains all static successor program points of a program point. It helps the illustration, since types at a program point are often related to those at static successors.

Program	LV	ST	MD	Succ's
`Method void m(int,int)`				
`.limit locals 5`				
`0 jsr 5`	$[c,int,int,uu,uu]$	$[]$	$[]$	5
`1 aload 1`	$[c,uu,int,uu,uu]$	$[]$	$[]$	2
`2 astore 3`	$[c,uu,int,uu,uu]$	$[c]$	$[]$	3
`3 jsr 5`	$[c,uu,c,uu,uu]$	$[]$	$[]$	5
`4 return`	$[c,uu,c,uu,uu]$	$[]$	$[]$	returns
`5 astore 4`	$[c,uu,uu,uu,uu]$	$[mks(5)]$	$[(5,\{\})]$	6
`6 aload 1`	$[c,uu,uu,mks(5),uu]$	$[]$	$[(5,\{4\})]$	7
`7 aload 1`	$[c,uu,uu,mks(5),uu]$	$[c]$	$[(5,\{4\})]$	8
`8 if_acmpeq 11`	$[c,uu,uu,mks(5),uu]$	$[c,c]$	$[(5,\{4\})]$	11,9
`9 aload 1`	$[c,uu,uu,mks(5),uu]$	$[]$	$[(5,\{4\})]$	10
`10 astore 2`	$[c,uu,uu,mks(5),uu]$	$[c]$	$[(5,\{4\})]$	11
`11 jsr 12`	$[c,uu,uu,mks(5),uu]$	$[]$	$[(5,\{2,4\})]$	12
`12 astore 5`	$[c,uu,uu,mks(5),uu]$	$[mks(12)]$	$[(5,\{2,4\}),(12,\{\})]$	13
`13 ret 4`	$[c,uu,uu,mks(5),mks(12)]$	$[]$	$[(5,\{2,4\}),(12,\{5\})]$	1,4

Figure 2: Types for the method in Figure 1

The types are c, int, $mks(pp)$ and uu, where c is class containing the method, int the type of all integers, $mks(pp)$ the type of all return program points of the subroutine pp, and uu the type of all data. Since uu is the type of all data, including data of c, int and $mks(pp)$, memory locations with the type uu are unusable (by most instructions).

For the instruction `jsr 5` at the program point `0`, whch calls the subroutine 5, the program point 1 is the return program point of the type $mks(5)$.

These types together with an artificial element $\perp$ build a lattice, whose partial ordering $\sqsupseteq$ is defined by that $uu \sqsupseteq c \sqsupseteq \perp$, $uu \sqsupseteq int \sqsupseteq \perp$ and $uu \sqsupseteq mks(pp) \sqsupseteq \perp$ for all program points pp. For any types ty and ty', we say that *ty covers ty'*, if $ty \sqsupseteq ty'$.

Now we may check the soundness property by flow analysis. For example, at the program point `0`, the first three local variables will hold the `this`-object and the two actual parameters, respectively, thus their assigned types should be c, int and int, respectively. The last two local variables at `0` may contain any data, thus their types should be uu. The instruction `jsr 5` at `0` and `3` will push its return program point onto the operand stack and transfer control to `5`, thus local variables at `5` should have types covering those at `0` and `3`, and the type of the stack entry at `5` should be the type $mks(5)$, and so on.

The above analysis goes quite smoothly until we encounter the instruction **`ret 4`** at `13`. Basically, the instruction **`ret 4`** needs three special treatments.

First, the control flow, i.e. the successor program points, of the instruction depend on the type assigned to local variable `4`. Since the assigned type is $mks(5)$, the successors are the return program points of the subroutine `5`, i.e. the program points `1` and `4`.

Second, no subroutine types for the subroutine `5` and all its inner subroutines should appear outside the subroutine `5`. Thus the subroutine types $mks(5)$ and $mks(12)$ for local variables `4` and `5` at `13` are replaced by type uu at `1` and `4`.

To understand the third special treatment, we need to consider a design decision made in the OJVMS, which enables a local variable that is not modified (by an **`astore`** instruction) in a subroutine, for example local variable `3`, to remain usable after each call of the subroutine. But the problem is that local variable `3` has different types int and c at different calling sites `0` and `3`, and thus should be assigned with the type uu in the subroutine `5`, covering different types at the calling sites. Now if we followed a naive approach by assigning the type uu to local variable `3` after the subroutine, then local variable `3` would practically become unusable. In other words, if we inserted e.g. **`aload 2`** at `1` or `4`, then the method would become illegal.

To overcome the above difficulty, the OJVMS requires that if a local variable is impossible to be modified in a subroutine, then at each return program point, it should be assigned with a type covering that at the corresponding call site. This leads to the need of *modification histories* to record modified local variables.

In Figure 2, modification histories are given under the column *MD*. Technically, a modification history contains a set of local variables for each enclosing subroutine such that the subroutine is the closest one containing modifications of these local variables. The set of all modified local variables for a given subroutine can then be computed as the union of the sets of local variables for all subroutines that the given subroutine encloses. For example, the modification history $[(\texttt{5}, \{2,4\}), (\texttt{12}, \{5\})]$ at `13` records that the subroutine `5` is the closest one containing modifications of local variables `2` and `4`, and that the subroutine `12` the closest one containing modifications of local variable `5`. The set of all modified local variables for the subroutine `5` is the union $\{2,4\} \cup \{5\}$.

The third special treatment for **`ret 4`** is in fact a way to compute the types at each return program point using the information on modified local variables of the subroutine and the types both at the calling **`jsr`** and **`ret`** instruction of the subroutine. In the above example, we use the modification history at `13` to compute the set $\{2,4,5\}$ of all modified local variables in the subroutine `5`. The assigned types for all modified local variables at `1` and `4` are in fact those at `13`. The assigned types for other local variables, i.e. local variables `1` and `3`, at `1` and `4` are those at `0` and `3`, respectively. This last point is in fact invisible for local variable `1`, since its type is c everywhere. But it is indeed visible for local variable `3`, where its type at `1` and `4` is int and c, respectively, the same as its type at `0` and `3`, respectively, independent of its type uu at `13`.

3 The problem and overview of our result

Based on discussion in the previous section, it does not seem difficult to write a non-deterministic fixed-point iteration algorithm, which repeatedly visits program points in some way, checks types at each program point against the constraints enforced by a typing rule, and computes the least types bigger the current ones at each successor so that the constraints enforced by the typing rule at that successor are satisfied. The algorithm is executed until all constraints are satisfied, or a failure is reported. The question is, however, whether the algorithm is terminating, computes types satisfying all constraints if assigning types is possible, and reports a failure if impossible.

Unfortunately, the algorithm is not entirely a standard non-deterministic fixed-point iteration. In particular, the algorithm is not monotone.

To demonstrate the situations, let us consider the method in Figure 3. We informally call a collection of all assigned types and modification histories a *type*. Let us first look at how the method types mty_1 and mty'_1 are obtained. Actually if the algorithm visits the method head and the program points `0`, `1`, `2`, `3`, `5`, `7`, `8`, `9`, `10` in that order, then it will compute the type *int* for local variable 2 at the successors program points `0`, `1`, `2`, `5`, `7`, `8`, `9`, `10`, `11` and `13`. Now if the algorithm visits the "else"-branch of the first **`if_acmpeq`**, i.e. the program points `3` and `4`, then it will change the type *int* for local variable 2 at `5` into *uu*, since local variable 2 at `5` may hold an integer or object. Since the algorithm has not visited the "else"-branch of the second **`if_acmpeq`**, i.e. `11` and `12`, the modification history $[(7, \{\})]$ at `13` does not record that local variable 2 is modified in the subroutine `7`. Now we have the method type mty_1. If the algorithm continues to visit the program point `13`, then it will take the type *uu* for local variable 2 at `5` as the type at the return program point `6`. Now we have the method type mty'_1.

	mty_1 and mty'_1			mty_2 and mty'_2		
Code	*VT*	*KT*	*MD*	*VT*	*KT*	*MD*
`Method void n(int)`						
`.limit locals 3`						
`0 aload 1`	$[c, int, uu]$	$[]$	$[]$	_	_	_
`1 aload 1`	$[c, int, uu]$	$[c]$	$[]$	_	_	_
`2 if_acmpeq 5`	$[c, int, uu]$	$[c, c]$	$[]$	_	_	_
`3 aload 1`	$[c, int, uu]$	$[]$	$[]$	_	_	_
`4 astore 2`	$[c, int, uu]$	$[c]$	$[]$	_	_	_
`5 jsr 7`	$[c, uu, uu]$	$[]$	$[]$	_	_	_
`6 return`	$[c, \underline{uu}, uu]$	$[]$	$[]$	$[c, \underline{int}, uu]$	_	_
`7 astore 3`	$[c, int, uu]$	$[st(7)]$	$[(7, \{\})]$	_	_	_
`8 aload 1`	$[c, int, st(7)]$	$[]$	$[(7, \{\})]$	_	_	_
`9 aload 1`	$[c, int, st(7)]$	$[c]$	$[(7, \{\})]$	_	_	_
`10 if_acmpeq 13`	$[c, int, st(7)]$	$[c, c]$	$[(7, \{\})]$	_	_	_
`11 iload 2`	$[c, int, st(7)]$	$[]$	$[(7, \{\})]$	_	_	_
`12 istore 2`				$[c, int, st(7)]$	$[int]$	$[(7, \{\})]$
`13 ret 3`	$[c, int, st(7)]$	$[]$	$[(7, \{\})]$	_	_	$[(7, \{2\})]$

Figure 3: Analyzing another JVM program

If starting from the method type mty_1 in the above, the algorithm continues to visit the "else"-branch of the second **`if_acmpeq`**, then we will arrive the method type mty_2, where the modification history at `13` is changed into $[(7, \{2\})]$, recording that local variable 2 is modified in the subroutine `7`. Note that the notation _ in the figure means that the type or modification history is the same as in mty_1. Now if the algorithm visits `13`, then it will take the type *int* for local variable 2 at `13` as the type at `6`. The final stage is the method type mty'_2.

Now it is easy to see that mty_1 is covered by mty_2 in the sense that each type or modification history in mty_1 is covered by the corresponding one in mty_2, whereas after an analysis step at `13`, mty'_1 is not covered by mty'_2, since the type *uu* for local variable 2 at `6` in mty'_1 strictly covers the type *int* for local variable 2 at `6` in mty'_2. This shows intuitively that the algorithm is not monotone.

The problem lies in that the type *uu* for local variable 2 at `5` in mty_1 is not covered by the type *int* for local variable 2 at `13` in mty_2. We discovered that this problem would not exist if mty_2 were a fixed point. The reason is that an analysis step actually always ensures that if a local variable is not modified, then its type at the current program point is covered by that at a successor program point. Thus if a method type is a fixed point, then for a path on which a local variable is not modified, its type at the beginning program point is always covered by that at the ending program point. Therefore, if mty_2 were a fixed point, then the type for local variable 2 at `5` would be covered by that at `13`. Since the type for local variable 2 at `5` in mty_1 is covered by the corresponding one in mty_2, the type for local variable 2 at `5` in mty_1 would be covered by that at `13` in mty_2.

Fortunately, the monotonicity property for method types where the bigger one is a fixed point suffices for the completeness of our algorithm. In fact, using this property and an additional property that the algorithm always produces a finite strictly-increasing chain of method types, we can prove that the algorithm will definitely reach the least solution if there are any.

To conclude this section, let us mention a temptation and dismiss it. The temptation is due to that the problem in the above seems also to lie in that there are paths with the same beginning and ending program points, where some do and some do not modify the same local variable. One may ask why we don't try to first finish derivation of all modification histories before starting the computation of types. The answer is that it is in general impossible to derive all entire modification histories before computing at least some types. In fact, in order to compute all modification histories, we need to consider all possible execution paths. But an execution path may depend on successor program points of a `ret`, which in turn depend on the type we want to compute for the local variable that the `ret` uses.

4 Preliminaries

We use the notation $\overline{\alpha_n}$ to denote a sequence of 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 $0 \leq i \neq j \leq n$, to denote a (finite) mapping, which maps each α_i to α'_i. We define $\mathcal{D}om(\{\overline{\alpha_n \mapsto \alpha'_n}\}) = \{\overline{\alpha_n}\}$. For a mapping θ, we use $\theta(\alpha)$ to denote the result of the mapping for α. Note that θ is not a function. Actually $\theta(\alpha)$ denotes an application of a specific lookup function on two first-order terms θ and α. We use $\theta[\alpha \mapsto \alpha']$ to denote the mapping that is equal to θ except it maps α to α'. A *list* $[\overline{\alpha_n}]$ is a special mapping $\{\overline{n \mapsto \alpha_n}\}$. We define $[\overline{\alpha_n}] + \alpha = [\overline{\alpha_n}, \alpha]$.

In formalizing our specification, we chose to use a constraint-solving framework based on a first-order many-sorted order-sorted algebra, which consists of a collection of sets, called *sorts*, a subset relation among the sorts, *functions* and *predicates* on these sorts (cf. e.g. [9]). A function is uniquely determined by a name. A *predicate* is uniquely determined by a name and an *arity*.

There is a set of *variables* for each sort.

Term are always well-sorted, and a term of a sort is automatically a term of each of its supersorts. Predicates are applied in a well-sorted way. *Logical formulas* are built as usual.

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

For simplicity, we may omit the explicit definition of a sort, function, predicate and variable in the paper, if it is standard or clear from the context. We will use names composed of low-case letters to denote sorts, functions and predicates. We will in general use the sort name to range over all elements in the sort, and the partially capitalized version of a sort name, where only the first letter is a capital, to range over all terms of the sort. Occasionally we will also use additional symbols to range over elements for the sake of clarity. For example, we will use *pp* and *ty* to denote an element of the sort *pp* and *ty*, respectively, and *Pp* and *Ty* a term of these sorts, respectively. In addition, we will use *sb* to denote an element in *pp* for some special uses of program points.

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 $i = 1, \cdots, n$. We use σ to range over all variable assignments. Applying a variable assignment σ to a term or logical formula s yields a result denoted as $\sigma(s)$, where all free occurrences of $X \in \mathcal{D}om(\sigma)$ in s are replaced by $\sigma(X)$.

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

5 The instance method *Mth* and the types

In this paper, we consider an arbitrary but fixed instance method Mth. Formally Mth is a nonempty list of instructions, where the positions of instructions are called *program points*. We use $\mathcal{D}om(Mth)$ to denote the set of all program points. We define a function $head$ that yields the head of the method Mth with $head(Mth) = (cls, \overline{ty_m}, n)$, where cls is the class declaring the method Mth, $\overline{ty_m}$ are the formal parameters, n is the number of lcoal variables such that $m \leq n$.

We use pp and sb to range over all elements in $\mathcal{D}om(Mth)$, where sb is used to represent (the starting program point of) a subroutine. We use ix to range over the indices of all local variables, and cn over all class names.

5.1 Types

In this paper we will consider the following *types* for memory locations:

Object type:	$\{\overline{cn_n}\}$ $(n > 0)$	Primitive type:	int
Subroutine type:	$mks(pp)$	Unusable value type:	uu

Intuitively, an *object type* is a class set, indicating that a memory location may only hold objects of the classes in the set. For this purpose, using a class set has the same effect as using the least common superclass of all classes of that set with respect to a subtyping relation. Thus we need not explicitly consider a subtyping relation any more in this paper. For convenience, we regard a singleton object type $\{cn\}$ and its class cn as the same.

The type int corresponds to the JVM primitive type int.

The type $mks(pp)$ intuitively indicates that a memory location may only hold return program points of the subroutine pp, i.e. the next program points of all instructions of the form `jsr` pp. We regard $mks(pp)$ and $mks(pp')$ with $pp \neq pp'$ as distinct. For example, the program points `1` and `4` in Figure 1 are of the type $mks(5)$, and the program point `12` is of the type $mks(12)$.

The type uu intuitively indicates that a memory location may hold all data. Such a memory location is unusable (by most instructions)[1].

We define some sorts in terms of the following syntax, where the non-terminals on the left of ::= are sorts.

Object type	ot	::=	$\{\overline{cn_n}\}$ $(n > 0)$
Subroutine type	st	::=	$mks(pp)$
Object or subroutine type	os	::=	$ot \mid st$
Type	ty	::=	$int \mid os \mid uu \mid \perp$

The subsort relation on these sorts is defined as the subset relation. For example, ty is a supersort of os, os a supersort of ot and st, and so on. Furthermore, since a singleton object type $\{cn\}$ and its class cn are the same, ot is a supersort of cn.

Note that we introduce an artificial bottom element $\perp$ in the sort ty.

We define a partial relation $\sqsupseteq$ for the sort ty as follows: for any ty and ty', $ty \sqsupseteq ty'$ if and only if (1) $ty = uu$, (2) $ty' = \perp$, or (3) ty and ty' are both object types and $ty \supseteq ty'$. Intuitively, for a memory location that may hold data of two types, if not both types are classes, then the memory location should have the

[1] In the full JVM, some stack manipulation instructions like `pop` can handle a memory location with the type uu.

type uu; if both types are classes, then the memory location should have the common superclass of these two classes (i.e. the set containing both classes) as its type.

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

5.2 Complex types and variable assignments

We construct the following sorts for complex types:

$$\begin{array}{llcl}
\text{Type list} & lst & ::= & [\overline{ty_n}]\ (n \geq 0) \mid \bot \mid \top \\
\text{Index set} & ixs & ::= & [\overline{ix_n}]\ (n \geq 0) \\
\text{Modification history} & mod & ::= & [\overline{(sb_n, ixs_n)}]\ (n \geq 0, sb_i \neq sb_j \text{ for } i \neq j) \\
\text{Program point type} & pty & ::= & (lst, lst', mod) \mid \bot \mid \top \\
\text{Method type} & mty & ::= & \{pp \mapsto pty_{pp} \mid pp \in \mathcal{D}om(Mth)\} \mid \bot \mid \top
\end{array}$$

The types of all local variables as a whole can be described a *type list*. The same holds for the types of all stack entries in a stack.

Modification histories are used to record local variables modified (by an `astore` instruction) in subroutines. Intuitively, a modification history $[\overline{(sb_n, ixs_n)}]$ should represent the following:

- The subroutines $\overline{sb_n}$ should be called (but not returned) in that order from the start of the method Mth to the current program point.
- Each set ixs_i with $1 \leq i < n$ should consist of the indices of all those local variables modified from sb_i to sb_{i+1}, the set ixs_n all those modified from sb_n to the current program point.

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 represents that a subroutine calls another one, 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, we choose to consider a simplified case, where only a single list of called subroutines is allowed.

A *program point type* (lst, lst', mod) records a list lst for all types of local variables, a list lst' for all types of stack entries, and a modification history mod.

A *method type* $\{pp \mapsto pty_{pp} \mid pp \in \mathcal{D}om(Mth)\}$ is defined to record all program point types in Mth.

For each of the sorts lst, mod, pty and mty, we define a partial relation $\sqsupseteq$ that componentwise extends the $\sqsupseteq$ on the sort ty and the subset relation on the sort ixs in the standard way. In particular, the following hold.

- $\bot$ and $\top$ are the artificial bottom and top in lst, pty and mty.
- All elements containing an occurrence of a $\bot$ (or $\top$) are regarded as identical to $\bot$ (or $\top$, respectively),
- $[\overline{ty_n}] \sqsupseteq [\overline{ty'_m}]$ if and only if $n = m$ and $ty_i \sqsupseteq ty'_i$ for all $i = 1, \cdots, n$.
- $[\overline{(sb_n, ixs_n)}] \sqsupseteq [\overline{(sb'_m, ixs'_m)}]$ if and only if $n = m$ and $sb_i = sb'_i$ and $ixs_i \supseteq ixs'_i$ for all $i = 1, \cdots, n$.

For each of the sorts lst, mod, pty and mty, we can also define the meet and join operations $\sqcap$ and $\sqcup$. Therefore, these sorts are all finite lattices.

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 for all $X \in \mathcal{D}om(\sigma_1)$,

$$\begin{array}{ll}
\sigma_1(X) \sqsupseteq \sigma_2(X) & \text{if } \sqsupseteq \text{ is defined for all elements in the sort of } X \\
\sigma_1(X) = \sigma_2(X) & \text{otherwise}
\end{array}$$

and define $\sigma_1 \sqcap \sigma_2$ by that $\mathcal{D}om(\sigma_1 \sqcap \sigma_2) := \mathcal{D}om(\sigma_1)$ and

$$\sigma_1 \sqcap \sigma_2(X) := \begin{cases} \sigma_1(X) \sqcap \sigma_2(X) & \text{if } \sqsupseteq \text{ is defined for all elements in the sort of } X \\ \sigma_1(X) & \text{otherwise} \end{cases}$$

6 Typing rules

We use variables of sorts as given in the following table

Variables	P, S	VT, KT	I	O	Y	MD	Φ
Sort	pp	lst	ix	ot	os	mod	mty

where S typically stand for a subroutine, VT for types of all local variables and KT for types of all stack entries.

A typing rule in our paper is in the form:

$$\frac{\begin{array}{c}\boxed{\mathcal{AC}}\\ \mathcal{CC}\end{array}}{\mathcal{SC}}$$

where $\mathcal{AC}$, $\mathcal{CC}$ and $\mathcal{SC}$ are sets of constraints.

We distinguish between these three sets in a typing rule, since they are used in three different places in an analysis process. Roughly speaking, the constraints in $\mathcal{AC}$ should determine a program point pp of application, thus are called *applicability conditions*. The $\mathcal{CC}$ should contain constraints on the program point type at the determined pp. The $\mathcal{SC}$ should contain constraints, if any, on the program point type at successors of the determined pp.

Independent of any analysis process, the typing rules as a whole enforce constraints on one common method type Φ of Mth. Let $Q = \mathcal{FV}(\mathcal{AC}) - \{\Phi\}$ and $Q' = \mathcal{FV}(\mathcal{CC} \cup \mathcal{SC}) - (\{\Phi\} \cup Q)$. Then a typing rule formally denotes the constraint

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

We say that *Mth has a method type mty*, (or *mty is a method type* of *Mth*,) if and only if the union of the constraints of all typing rules are satisfied under the same variable assignment $\{\Phi \mapsto mty\}$. Note that the variable Φ is global in all typing rules. The method Mth may have zero or more method type.

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

We are ready to give typing rules now. For notational simplicity, we assume the following:

- In all typing rules except rule (1), the $\mathcal{AC}$ part always implicitly contains the constraint $\Phi(P) \neq \bot$.
- In all typing rules, the $\mathcal{CC}$ part always implicitly contains all necessary checks ensuring e.g. that terms of the sort pp are in $\mathcal{D}om(Mth)$, and that terms of the sort ix are between 1 and n, etc.

Figure 4 contains the first part of our typing rules. Rule (1) is always applicable since the $\mathcal{AC}$ part is an empty set. Remember that we have assumed that $head(Mth) = (cls, \overline{ty_m}, n)$. In Figure 2, we have that $cls = c$, $ty_1 = int$, $ty_2 = int$, $m = 2$, and $n = 5$.

Rules (3) and (5) use the following auxiliary function to add a set of indices of local variables to the set of modified local variable of the current subroutine:

$$addmlvs(ixs, \overline{(sb_n, ixs_n)}) = \begin{cases} [\overline{(sb_{n-1}, ixs_{n-1})}] + (sb_n, ixs_n \cup ixs) & \text{if } n \geq 1 \\ [] & \text{if } n = 0 \end{cases}$$

In Figure 2, rule (3) is applicable at the program point 2 with $P = 2$ and $I = 3$. The $\mathcal{CC}$ and $\mathcal{SC}$ are satisfied at 2 with $VT = [c, uu, int, uu, uu]$, $KT = []$, $Y = c$ and $MD = []$. Note that $addmlvs(\{3\}, []) = []$. Rule (3) is also applicable at the program point 5 with $P = 5$ and $I = 4$. The $\mathcal{CC}$ is satisfied with $Y = mks(5)$.

Note that both $Y = c$ and $Y = mks(5)$ in the above are well-sorted since both ot and st are subsorts of os. It is worth noticing that the variable O in rule (2) cannot be instantiated into an element of the sort st.

Rule (7) introduces no successor program points. Note that a method body must contain a **`return`** instruction.

$$\frac{\Box}{\Phi(0) \sqsupseteq (uu^n[0 \mapsto cls, \overline{m \mapsto ty_m}], [], [])} \tag{1}$$

$$\frac{\boxed{Mth(P) = \texttt{aload}\ I} \quad \Phi(P) = (VT, KT, MD) \quad VT(I) = O}{\Phi(P+1) \sqsupseteq (VT, KT + O, MD)} \tag{2}$$

$$\frac{\boxed{Mth(P) = \texttt{astore}\ I} \quad \Phi(P) = (VT, KT + Y, MD)}{\Phi(P+1) \sqsupseteq (VT[I \mapsto Y], KT, addmlvs(\{I\}, MD))} \tag{3}$$

$$\frac{\boxed{Mth(P) = \texttt{iload}\ I} \quad \Phi(P) = (VT, KT, MD) \quad VT(I) = int}{\Phi(P+1) \sqsupseteq (VT, KT + int, MD)} \tag{4}$$

$$\frac{\boxed{Mth(P) = \texttt{istore}\ I} \quad \Phi(P) = (VT, KT + int, MD)}{\Phi(P+1) \sqsupseteq (VT[I \mapsto int], KT, addmlvs(\{I\}, MD))} \tag{5}$$

$$\frac{\boxed{Mth(P) = \texttt{if\_acmpeq}\ P'} \quad \Phi(P) = (VT, KT + O + O', MD)}{\Phi(P') \sqsupseteq (VT, KT, MD) \quad \Phi(P+1) \sqsupseteq (VT, KT, MD)} \tag{6}$$

$$\frac{\boxed{Mth(P) = \texttt{return}}}{} \tag{7}$$

Figure 4: Typing rules

The second part of the typing rules are in Figure 5 and for subroutines.

In Figure 2, rule (8) is applicable at the program point `0` with $P = 0$ and $S = 5$ and also at the program point `3` with $P = 3$ and $S = 5$. The constraint $S \notin \mathcal{D}om(MD)$ ensures that no subroutines may be called recursively. The $\mathcal{SC}$ ensures that the type $mks(5)$ is pushed onto the stack at the successor `5` and the modification history is extended with $(5, \{\})$.

Rules (9) and (10) are both for `ret`. They are complementary: rule (9) focuses on constraints at the current program point, whereas rule (10) those at successor program points, i.e. return program points of the subroutine. We could combine them into one, but the result would be too clumsy.

In Figure 2, rule (9) is applicable at the program point `13` with $P = 13$ and $I = 4$. The constraint $VT(I) = mks(\_)$ ensures that local variable `4` at `13` has a subroutine type, and thus the local variable holds a return program point. The constraint $\forall P' \forall I' \forall VT'. \cdots$ ensures that no subroutines have multiple exits. This is not a serious restriction on programs, since multiple exits can always be changed into single exit using `if_acmpeq` (or `goto` in the full JVM).

Rule (10) has a relatively large $\mathcal{AC}$-part. If it is applicable under a variable assignment, then the variable assignment contains the information about the subroutine and a calling `jsr`. In Figure 2, rule (T-10) is applicable at the program point `13` with $P = 13$, $I = 4$, $S = 5$ and $P' = 0$ or $P' = 3$. Thus the successor program point is either $P' + 1 = 1$ or $P' + 1 = 4$.

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{jsr}\ S} \\ \Phi(P) = (\,VT, KT, MD) \\ S \notin \mathcal{D}om(MD)\end{array}}{\Phi(S) \sqsupseteq (\,VT, KT + mks(S), MD + (S, \{\}))} \tag{8}$$

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{ret}\ I} \\ \Phi(P) = (\,VT, \_, \_) \\ VT(I) = mks(\_) \\ \forall P' \forall I' \forall VT'.(\ (Mth(P') {=} \texttt{ret}\ I' \ \wedge\ P' {\neq} P \ \wedge\ \Phi(P') {=} (\,VT', \_, \_)\ ) \\ \qquad\qquad\qquad \Rightarrow VT(I) {\neq} VT'(I')\ )\end{array}}{} \tag{9}$$

$$\frac{\boxed{\begin{array}{l}Mth(P) = \texttt{ret}\ I \\ \Phi(P) = (\,VT, KT, MD) \\ Mth(P') = \texttt{jsr}\ S \\ VT(I) = mks(S) \\ \Phi(P') = (\,VT', KT', MD')\end{array}}}{\begin{array}{l}\Phi(P'{+}1) \sqsupseteq (\,VT'[j \mapsto chuu(\,VT(j), sbsin(MD,S)) \mid j \in mlvsin(MD,S)], \\ \qquad\qquad chuulis(KT, sbsin(MD, S)), \\ \qquad\qquad addmlvs(mlvsin(MD, S), modbef(MD, S))\end{array}} \tag{10}$$

Figure 5: Typing rules

The $\mathcal{SC}$ part of rule (10) uses the auxiliary function *addmlvs* and several other auxiliary functions. Let $[\overline{(sb_n, ixs_n)}]$ be a modification history, sb_k a subroutine with $1 \leq k \leq n$, E a set of subroutines, ty an arbitrary type, and $[\overline{ty_m}]$ a list of types. Then these functions are defined as follows:

$$\begin{aligned}
sbsin([\overline{(sb_n, ixs_n)}], sb_k) &= \{sb_k, \cdots, sb_n\} \\
mlvsin([\overline{(sb_n, ixs_n)}], sb_k) &= ixs_k \cup \cdots \cup ixs_n \\
modbef([\overline{(sb_n, ixs_n)}], sb_k) &= [\overline{(sb_{k-1}, ixs_{k-1})}] \\
chuu(ty, E) &= \begin{cases} uu & \text{if } ty = mks(sb) \text{ and } sb \in E \\ ty & \text{otherwise} \end{cases} \\
chuulis([\overline{ty_m}], E) &= [\overline{chuu(ty_m, E)}]
\end{aligned}$$

If rule (T-10) is applicable at `13` as described above, then the modification history at `13` is $MD = [(5, \{2{,}4\}), (12, \{5\})]$. Thus $sbsin(MD{,}S) = \{5, 12\}$ is the set of all subroutines called since S including S, $mlvsin(MD, S) = \{2{,}4{,}5\}$ the set of all local variables modified since S, and $modbef(MD, S) = []$ the part of modification history before calling the subroutine S. The modification history at the return program point $P' + 1 = 1$ or $P' + 1 = 4$ is $addmlvs(mlvsin(MD, S), modbef(MD, S)) = []$, which is obtained by adding $mlvsin(MD, S)$ to the current subroutine in $modbef(MD, S)$.

Furthermore, the functions $chuu(ty, E)$ and $chuulis(\overline{[ty_m]}, E)$ change the subroutine type ty and each subroutine type ty_i into the type uu if the subroutine type is for a subroutine in E. They serve to ensure that no return program points for the subroutine 5 and all those subroutines directly or indirectly called by the subroutine 5 are usable after the subroutine 5 returns. Actually, in the application described as above, we have that $chuu(VT(j), sbsin(MD,S)) = uu$ for $j = 2, 4$ or 5 and $chuulis(KT, sbsin(MD, S)) = []$

7 Computing a least method type or yielding a failure

Although our typing rules enforce constraints on possible method types of bytecode, they do not directly tell us how to compute a method type or whether there is a least method type among all method types.

We follow the idea in [11], which in turns follows the Cousots' approach to abstract interpretation [1], to write a non-deterministic (chaotic) fixed-point iteration algorithm. Roughly, the algorithm starts with the bottom method type $\perp$, repeatedly calls a procedure to compute a sequence of strictly-increasing method types until no new method types can be yielded, or a failure is reported by the procedure. The procedure always takes the last obtained method type, non-deterministically chooses a current program point and a typing rule applicable at the program point, checks the constraints enforced by the rule against the types at the current program point, attempts to compute a least method type that bigger than or equal to the last method type and satisfy the constraints enforced by the rule at each successor of the current program point. A failure will be yielded if the constraints at the current program point are not satisfied, or the yielded method type is $\top$.

From now on, when discussing a particular typing rule, we can always use the general notations given at the beginning of Section 6 and the special notations in the formulation of the typing rule without explicitly saying.

For formulating the algorithm, we need the following formal definitions.

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.

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

1. Let $mty_0 := \perp$.
2. Assume that mty_k is the last obtained intermediate method type. Choose a typing rule and a variable assignment σ such that $\sigma(\Phi) = mty_k$ holds and the rule is applicable under σ. Let *apply_a_rule* take the rule and σ as arguments.
   (a) If *apply_a_rule* yields a failure, then the algorithm terminates with a failure;
   (b) otherwise, if *apply_a_rule* yields a method type different than mty_k, then let mty_{k+1} be the yielded method type.
3. Repeat the above step with $k := k+1$, until the algorithm terminates with a failure, or *apply_a_rule* does not yield a new intermediate method type for every typing rule and variable assignment chosen as above. The algorithm terminates with either a failure or the last computed intermediate method type.

Figure 6: The algorithm *analyzer*

We formulate our algorithm *analyzer* in Figure 6. It produces a sequence of *intermediate method types* mty_0, $mty_2, \cdots$, and finally terminates with either a method type of the method *Mth* or a failure.

Input: A typing rule and σ such that the typing rule is applicable under σ.
Output: A method type or failure.
Body:

1. If $find\_a\_subst$ yields a variable assignment σ' for the given typing rule and σ, and if the method type

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

   is not $\top$, then the current procedure yields the method type;
2. otherwise, the current procedure yields a failure.

Figure 7: The procedure *apply_a_rule*

The algorithm *analyzer* calls the procedure *apply_a_rule* in Figure 7 to compute an intermediate method type. The procedure *apply_a_rule* takes a typing rule and a variable assignment σ as arguments, where $\sigma(\Phi)$ is the last obtained mty_k. The reason for the procedure to take an entire variable assignment σ, instead of the single method type mty_k, is that $\mathcal{AC}$ part of a typing rule may contain other variables than Φ, and that a variable assignment determines the assignment of these other variables as well.

The procedure *apply_a_rule* calls the procedure $find\_a\_subst$ in Figure 8 to compute a variable assignment σ' such that $\sigma'(\Phi) \sqsupseteq mty_k$ and the typing rule is pre-satisfied under σ', or reports a failure if such a σ' does not exists. This implies that the procedure *apply_a_rule* checks the $\mathcal{CC}$ against the types at the current program point in the method type mty_k.

Note that the definition in Figure 8 is naive and inefficient. We could have written a more efficient one, but the proof would become complicated.

Input: A typing rule and 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 $\sqcap$ to all these variable assignments.

Figure 8: The procedure $find\_a\_subst$

After the procedure $find\_a\_subst$ returns, the procedure *apply_a_rule* attempts to compute a new method type mty_{k+1}, which should be bigger than mty_k, not be $\top$, and satisfy all constraints enforced by the typing rule.

Theorem 1. *(Correctness) If analyzer yields a method type mty, then Mth has the method type mty.*

Theorem 2. *(Termination) If all auxiliary functions in typing rules are terminating, then analyzer is terminating.*

Theorem 3. *(Completeness) If Mth has a method type mty, then analyzer will yield a method type mty′ of Mth with mty $\sqsupseteq$ mty′.*

Corollary 4. *(Least-type property) If Mth is statically well-typed, then analyzer will always terminate with the least method type of it.*

7.1 Example

Figure 9 describes a computation of the method type in Figure 2 using the algorithm *analyzer*. Each row in Figure 9 records a program point type at a program point in an intermediate method type, together with some information about the application of a typing rule. The column "Step" contains numbers indicating the order of applications, the column "Rule" the names of the applied typing rules and the column "Succ's" all static successor program points. An application of a typing rule 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 written. The last program point types for all program points are exactly those in Figure 2.

P	VT	KT	MD	Step	Rule	Succ.'s
`Method void m(int,int)`				1	(1)	0
0	$[c,int,int,uu,uu]$	$[]$	$[]$	2	(8)	5
1	$[c,uu,int,uu,uu]$	$[]$	$[]$	16	(2)	2
2	$[c,uu,int,uu,uu]$	$[c]$	$[]$	17	(3)	3
3	$[c,uu,c,uu,uu]$	$[]$	$[]$	18	(8)	5
4	$[c,uu,c,uu,uu]$	$[]$	$[]$	30	(7)	returns
5	$[c,int,int,uu,uu]$	$[mks(5)]$	$[(5,\{\})]$	3	(3)	6
	$[c,uu,uu,uu,uu]$			19	(3)	6
6	$[c,int,int,mks(5),uu]$	$[]$	$[(5,\{4\})]$	4	(2)	7
	$[c,uu,uu,mks(5),uu]$			20	(2)	7
7	$[c,int,int,mks(5),uu]$	$[c]$	$[(5,\{4\})]$	5	(2)	8
	$[c,uu,uu,mks(5),uu]$			21	(2)	8
8	$[c,int,int,mks(5),uu]$	$[c,c]$	$[(5,\{4\})]$	6	(6)	11,9
	$[c,uu,uu,mks(5),uu]$			22	(6)	11,9
9	$[c,int,int,mks(5),uu]$	$[]$	$[(5,\{4\})]$	10	(2)	10
	$[c,uu,uu,mks(5),uu]$			23	(2)	10
10	$[c,int,int,mks(5),uu]$	$[c]$	$[(5,\{4\})]$	11	(3)	11
	$[c,uu,uu,mks(5),uu]$			24	(3)	11
11	$[c,int,int,mks(5),uu]$	$[]$	$[(5,\{4\})]$	7	(8)	12
	$[c,uu,int,mks(5),uu]$		$[(5,\{2,4\})]$	12	(8)	12
	$[c,uu,uu,mks(5),uu]$			25	(8)	12
12	$[c,int,int,mks(5),uu]$	$[mks(12)]$	$[(5,\{4\}),(12,\{\})]$	8	(3)	13
	$[c,uu,int,mks(5),uu]$		$[(5,\{2,4\}),(12,\{\})]$	13	(3)	13
	$[c,uu,uu,mks(5),uu]$			26	(3)	13
13	$[c,int,int,mks(5),mks(12)]$	$[]$	$[(5,\{4\}),(12,\{5\})]$	9	(9)	
	$[c,uu,int,mks(5),mks(12)]$		$[(5,\{2,4\}),(12,\{5\})]$	14	(9)	
				15	(10)	1
	$[c,uu,uu,mks(5),mks(12)]$			27	(9)	
				28	(10)	4
				29	(10)	1

Figure 9: Computing the method type in Figure 1

The computation has 30 application steps. Step 1 applies rule (1) and computes local variable types $[c,int,int,uu,uu]$ and a stack type [] at `0`. Step 2 applies rule (8), checks the local variable types $[c,int,int,uu,uu]$ and the stack type [] at `0` and computes local variable types $[c,int,int,uu,uu]$ and a stack type $[mks(5)]$ at `5`, and so on.

After step 9, which applies rule (9), an application of rule (10) could be considered at `13`. But we choose first to continue at `9` to consider the "else"-branch of `if_acmeq`.

After step 14, which applies rule (9), we choose to consider an application of rule (10) at `13` as step 15. Now we have the modification history $mod = [(5,\{2,4\}),(12,\{5\})]$, thus $mlvsin(mod,5) = \{2,4,5\}$, $sbsin(mod,5) = \{5,12\}$ and $modbef(mod,5) = []$. Step 15 computes a program point type at `1`. Local variables 4 and 5 at `1` have the type uu, since they have the subroutines types $mks(5)$ and $mks(12)$ at `13`, where $4,5 \in mlvsin(mod,5)$ and $5,12 \in sbsin(mod,5)$ hold. Local variable 2 at `1` has the type uu, covering that at `13`, since $2 \in mlvsin(mod,5)$ holds. Local variables 1 and 3 at `1` have its types c and int at `0`, respectively, since $1,3 \notin mlvsin(mod,5)$.

Step 28 is an application of rule (10) at `13` with respect to the `jsr` at `3`, computing a new program point type at `4`. For the same reason as before, local variables 4 and 5 at `4` have the type uu, local variable 2 at `4` has the type uu, covering that at `13`, local variable 1 and in particular local variable 3 at `4` take their types c and c at `3`, since $1,3 \notin mlvsin(mod,5)$. Note that local variable 3 at `13` has the type uu: it does not directly affect its type at `4`.

8 Related work

Stata and Abadi [10] 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 [2] made a significant extension of Stata and Abadi's type system by considering object initialization. Qian [8] presented a constraint-based typing system for objects, primitive values, methods and subroutines and proved the soundness. Hagiya and Tozawa [4] presented another type system for subroutines, where the soundness proof is extremely simple. Pusch [7] formalized a subset of JVM in the theorem prover Isabelle/HOL and reached a higher degree of reliability. Recently, O'Callahan [6] proposed yet another typing system based on type constraints, polymorphic recursion and continuations. All this work basically aimed at defining what types memory locations should have, did not consider how to develop a provably-correct implementation to compute types for memory locations.

Hagiya and Tozawa [4] discussed issues on an implementation of their type system, but did not formally describe the implementation. In addition, since they did not consider objects, their implementation may not encounter the problems we have here.

Goldberg [3] directly used dataflow analysis to formally specify bytecode verification focusing on type-correctness and global type consistency for dynamic class loading. He gave transfer functions that correspond to the use of typing rules in our algorithm. But he did not give a logical interpretation for the transfer functions. In addition, since he did not consider subroutines, he did not encounter the problems we discuss in this paper.

Although all the approaches mentioned above look different and deal with different subsets of JVM instructions, they all, except [6], directly or indirectly handle flow analysis in some ways. Thus the correspondences are quite straightforward.

A previous version of the current paper proposed an algorithm where the application of rule (10) needs to be restricted. The current paper shows that the restriction can be removed.

In summary, to our knowledge, there has not been much work formally proving the existence of least types, formalizing algorithm that computes types to memory locations in Java bytecode and proving properties of the algorithm. Indeed, our proofs need to make use of the special properties of the `jsr` and `ret` instructions, which do not exist in other existing systems.

9 Conclusion

We have proposed a non-deterministic algorithm, which attempts to assign, to memory locations in bytecode, types satisfying a set of typing rules. We have formally proved that the algorithm is terminating, and yields the least assigned types if there are any, or reports a failure if there are none. We have implicitly proved the existence of least types for memory locations in bytecode if there are any.

Although it has been obtained in the context of a particular approach, we strongly conjecture that the result on the existence of least types can be carried over to other approaches that directly or indirectly handle flow analysis. (But we cannot make the conjecture for the approach in [6].)

Our algorithm directly uses the formulation of typing rules in its computation. In other approaches, there may be a gap between the typing rules and an algorithm that computes least types. As an example, consider the type system by Stata and Abadi [10]. In that system, types of local variables are formalized as a partial map from indices to types such that only those local variables that are allowed to be touched at a program point are included in the map for that program point. The typing rule for `jsr` non-deterministically decides which local variables are included in the partial maps within the subroutine. Thus it is difficult to use their typing rules to compute least types, though the typing rules may serve as a specification.

In practice, the existence of least types suggests that if bytecode verifiers are well designed and implemented, then a mobile code that can be accepted by one bytecode verifier can also be accepted by all bytecode verifiers. This implies that good bytecode verifiers will not reject services provided by legal bytecode.

Acknowledgements

We sincerely thank one anonymous referee for detailed and constructive comments on a previous version of this paper. The research is partially supported by DARPA contract F30602-96-C-0363.

References

1. 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.
2. S. Freund and J. Mitchell. A type system for object initialization in the java bytecode language. In *Proc. Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, pages 310–328. ACM Press, 1998.
3. A. Goldberg. A specification of Java loading and bytecode verification. In *Proc. 5th ACM Conference on Computer and Communications Security*, 1998.
4. 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.
5. T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification.* Addison-Wesley, 1996.
6. R. O'Callahan. A simple, comprehensive type system for Java bytecode subroutines. In *Proc. 26th ACM Symp. Principles of Programming Languages*, 1999. To appear.
7. 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/`. A short version in OOPSLA'98 Workshop "Formal Underpinnings of Java".
8. 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 1523, 1998.
9. 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.
10. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proc. 25th ACM Symp. Principles of Programming Languages*, 1998.
11. R. Wilhelm and D. Maurer. *Compiler Design.* Addison-Wesley Publishing Company, 1995.