A Formal Specification of Java™ Virtual Machine Instructions for Objects, Methods and Subroutines*

Zhenyu Qian

Bremen Institute for Safe Systems (BISS), FB3 Informatik, Universität Bremen, D-28334 Bremen, Germany

Abstract. In this chapter we formally specify a subset of Java Virtual Machine (JVM) instructions for objects, methods and subroutines based on the official JVM Specification, the official Java Language Specification and Sun's JDK 1.1.4 implementation of the JVM. Our formal specification describes the runtime behaviors of the instructions in relevant memory areas as state transitions and most structural and linking constraints on the instructions as a static typing system. The typing system includes a core of the Bytecode Verifier and resembles data-flow analysis. We state some properties based on our formal specification and sketch the proofs. One of these properties is that if a JVM program is statically well-typed with respect to the typing system, then the runtime data of the program will be type-correct. Our formal specification clarifies some ambiguities and incompleteness and removes some (in our view) unnecessary restrictions in the description of the official JVM Specification.

1 Introduction

The Java Virtual Machine (JVM) is a platform-independent abstract computing machine containing an instruction set and running on various memory areas. The JVM is typically used as an intermediate machine in the implementation of the programming language `Java`. The official JVM Specification by Lindholm and Yellin [10] (OJVMS) defines the syntax of the instructions and describes the semantics of the instructions in related memory areas.

This chapter specifies a subset of the instructions for objects, methods and subroutines by giving a formal semantics to them. The formal specification is based on the OJVMS, Sun's JDK 1.1.4 implementation of the JVM, in particular, the Bytecode Verifier, and the official Java Language Specification by Gosling, Joy and Steele [8] (OJLS). The formal specification provides a foundation for exposing the behaviors of the subset of the JVM. Since programs of the instructions in the JVM can be used directly over the Web, our formal specification defines parts of the security of internet programming in `Java`.

* To appear in Jim Alves-Foss (ed.) "Formal Syntax and Semantics of Java™", Springer Verlag LNCS, 1998

The formal specification considers the following essential instructions: the load and store instructions for objects and integers, the object creation instruction, one operand stack management instruction, several control transfer instructions, all method invocation instructions, several return instructions, and the `jsr` and `ret` instructions for implementing `finally`-clauses.

Many features in the JVM are not considered in this chapter. They are multi-threads, arrays, primitive types other than type *int*, two-word wide data, class initialization method `<clinit>`, access control modifiers, exception handlings, native methods, `lookupswitch`, `tableswitch`, `wide`, runtime exceptions, memory organization, the overflow and underflow of the operand stack, the legality of accesses of local variables, the `class` file format in details, constant pool resolution in details and the difference between "static" and "link time". We assume that all classes have been loaded by a single class loader. Due to space constraints we only very briefly sketch all proofs in this chapter.

The paper [12] considers a larger subset of JVM instructions, in particular, those for exception handling. In addition, it contains the proofs.

The main ideas of our approach are as follows:

– We formalize an operational semantics of the instructions by defining each instruction as a state transition.
– At the same time we formulate a static typing system. Based on the typing rules in the system, one may try to derive a static type for each memory location such that the static type covers the types of all runtime data possibly held by the memory location. The typing system characterizes aspects of the data-flow analysis (see e.g. [1]).
– Our formal specification consists of the state transition machine and the typing system. The state transition machine is defined only for programs, where static types for all memory locations are derivable with respect to the typing system. Practically, the typing system includes a core of the Bytecode Verifier.
– We finally state some properties of the formal specification. In particular, we state that if the type inference system can be successful, then the runtime data are guaranteed to be type-correct.

To a large extent, our formal specification follows the OJVMS. However, some extensions and changes of the semantics are necessary and desirable. Four of them are as follows:

– The OJVMS (page 130) requires that the static type of an operand stack entry or a local variable should be the least upper bound of the types of all possible runtime data in it, and the least upper bound should be one JVM type. The problem is, however, that the subtyping relation on interfaces allows multiple inheritance and thus two interfaces need not have one least common superinterface. Our solution is to allow a set of interfaces (and classes) to be a static type of an operand stack entry or a local variable.

– The OJVMS (page 132) uses a special type indicating that an object is *new*, i.e. it has been created by the instruction `new` but not yet initialized by an instance initialization method. We introduce two kinds of special types indicating two different stages of object initialization in the specification: one indicates that the object is uninitialized; the other indicates that the object is being initialized by an instance initialization method, but has not yet encountered the invocation of another instance initialization method. We distinguish between these two stages because the objects at different stages should be dealt with differently.
– The OJVMS introduces a concept of *subroutines*: a `jsr` instruction jumps to or calls a subroutine and a `ret` instruction *returns* from a subroutine. The mechanism of subroutines is based on the correct use of return addresses. The OJVMS defines a new primitive type `returnAddress` indicating that a value is a return address. For the formal specification we refine type `returnAddress` into a family of special types, called *subroutine types*, where a value of a subroutine type is the address of a `jsr` instruction calling a subroutine and thus can be used to compute the return address of the subroutine. As we will see, subroutine types are crucial in our specification of constraints on `jsr` and `ret` instructions.
– The OJVMS does not clearly distinguish between types for memory locations and types for runtime data. Our formal specification clearly distinguish between static types for memory locations and types (or tags) of runtime data. Therefore, we can formally discuss the type safety property of runtime data in the execution.

In this chapter we use the following notations.

We use the notation $\overline{\alpha_n}$ to denote n syntactical objects $\alpha_1, \cdots, \alpha_n$, the notation $\{\cdots\}$ a set and define $size(\{\overline{\alpha_n}\}) := n$.

We use $\{\overline{\alpha_n \mapsto \alpha'_n}\}$, where $\alpha_i \neq \alpha_j$ hold for all i, j with $0 \leq i, j \leq n$ and $i \neq j$, to denote a mapping, where the mapping of each α_i is α'_i, and the mapping of every other element will be defined in each concrete case. In fact, in each concrete case, the mapping of every other element will always be either the element itself, or a special value *failure*, or not explicitly defined because it is never used. 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, and write $\theta[\alpha \mapsto \alpha']$ for the mapping that is equal to θ except it maps α to α'. For a set D, we define

$$\begin{aligned}\theta_{|D} &:= \{\alpha \mapsto \theta(\alpha) \mid \alpha \in \mathcal{D}om(\theta) \cap D\}\\ \theta_{|-D} &:= \{\alpha \mapsto \theta(\alpha) \mid \alpha \in \mathcal{D}om(\theta) - D\}\end{aligned}$$

A *list* $[\alpha_0 \cdots, \alpha_n]$ with $n \geq -1$ is a special mapping $\{i \mapsto \alpha_i \mid 0 \leq i \leq n\}$. For any list lis, we define $lis + \alpha := lis[size(lis) \mapsto \alpha]$.

2 Related work

Stata and Abadi proposed a type system for a set of instructions focusing on subroutines and proved the soundness [15]. Since they considered only a few instructions, they could provide lengthy proofs and clarify several key semantic issues about subroutines. Freund and Mitchell made a significant extension of Stata and Abadi's type system by considering object initialization [5,6], and in doing this, discovered a bug in Sun's implementation of the bytecode verifier, which allows a program to use an object before it has been initialized. To fix the bug, they wrote a typing rule that ensures that at no time during program execution, there may be more than one uninitialized object that is created by the same `new` instruction and is usable.

After realizing the bug discovered Freund and Mitchell, we detected that an early version of the current paper contains the same bug. Except for this point, the results of the current paper are independent of those by Stata, Abadi, Freund and Mitchell. There are several differences between our approach and theirs. First, we follow the constraint-solving framework and use typing rules to generate constraints that define all legal types. Second, we consider more JVM instructions and more details. Two examples are that our approach allows an inner subroutine to return directly to an outer `jsr` instruction in nested subroutines, whereas their approach does not, and that upon a subroutine return, our approach assigns a type to a local variable using the information on whether the local variable is modified in the subroutine, whereas their approach does not consider the case.

Cohen described a formal model of a subset of the JVM, called *defensive JVM* (dJVM), where runtime checks are used to assure type-safe execution [2]. Our approach is different in that we design a static type inference system, which assures that statically well-typed programs do not have runtime type errors. In addition, the current dJVM does not consider subroutines, whereas our specification does.

Goldberg gave a formal specification of bytecode verification [7]. Compared with our work, he considered array types, but not subroutines. In addition, his formal specification is a dataflow analysis and thus closer to the implementation.

Hagiya presented another type system for subroutines [9]. One of the interesting points in his approach is to introduce a mechanism to distinguish the so-called "used" from the "unused" data in a subroutine. His idea is to use a kind of special types indicating that a certain memory location in a subroutine always has the same content as a memory location at a call to the subroutine.

The Kimera project is quite successful in testing some running bytecode verifiers and detecting some flaws [14]. In general, testing is often based on a precise specification. Thus a formal specification may be useful for testing.

Dean [3] studied a formal model relating static typing and dynamic linking and proved the safety of dynamic linking with respect to static typing. As mentioned before, our formal specification does not consider the issue between static typing and dynamic linking.

Our formal specification considers only one single class loader. Saraswat studied static type-(un)safety in `Java` in the presence of more than one class loader [13] .

Although the JVM uses some structures of the `Java` language, our type system for the JVM resembles data-flow analysis and thus is quite different from a formal specification of a type system for `Java` in e.g. [4,11,16].

3 JVM programs, methods, data areas and frames

According to the OJVMS, a byte is 8 bits, and a word is an abstract size that is larger than, among others, a byte. One-byte-wide data build instructions, whereas one-word wide data represent runtime data. We use *byt* to range over all one-byte data and *wrd* over all one-word wide data.

The OJVMS still allows two-word wide integers. But, as mentioned before, we do not consider two-word wide data in this chapter for simplicity.

A *(JVM) program* in this chapter is defined to contain a set of *methods*. We assume that each method has a unique *method code reference*. We use *cod* to range over all method code references. An *address* is a pair (cod, off), where off is a one-word wide datum, called a *byte offset*. For any address (cod, off) and another byte offset off', we define $(cod, off) + off' := (cod, off + off)$. An instruction may be longer than one byte. A *program point*, denoted by *pp*, is the starting address of an instruction. Since we do not consider multi-threads, we assume that there is just one *program count register*, which contains the current program point.

As mentioned in the introduction to this chapter, the program point of a `jsr` instruction may be used in computing the returning program point for a subroutine. In fact, it is the byte offset of the program point, not the program point itself, that may be used, since, as we will see later, a `ret` instruction, which uses the program point of a `jsr` instruction, is always in the same method as the `jsr` instruction. Thus we may talk about *the byte offset* of a `jsr` in the rest of this chapter.

We consider an arbitrary but fixed program *Prg*. Note that the methods in a program may stem from different `class` files. A method in *Prg* consists of all instructions in *Prg* whose program points contain the same given method code reference. We use *Mth* to denote an arbitrary but fixed method in *Prg*.

We use $allPP(Prg)$ and $allPP(Mth)$ to denote the sets of all program points in *Prg* and *Mth*, respectively. We assume that $allPP(Mth)$ always contains one unique element of the form $(\_, 0)$. Intuitively, it is the starting program point of the method.

We define that the function $offset(byt1, byt2)$ yields $(byt1 * (2^8)) + byt2$ if it is a one-word wide value, a failure otherwise.

In our specification an *object reference* is formally a one-word wide datum. We use *obj* to range over all object references. Furthermore, we use *null* to denote a special object reference.

Following the OJVMS, we formally specify *int* as the primitive type of all one-word wide integers and use *val* to range over these integers.

We use *cnam*, *inam*, *mnam* and *fnam* to range over names of classes, interfaces, methods and fields, respectively. For our formal specification we require that *fnam* is always a qualified name.

A *record* is formally a mapping of the form $\{\overline{fnam_n \mapsto wrd_n}\}$, which maps all elements other than $\overline{fnam_n}$ to a special value *failure*. We use *rec* to range over all records.

The JVM has a *heap*, from which memory for all objects is allocated. Formally, a *heap state* is defined as a mapping of the form $\{\overline{obj_n \mapsto rec_n}\}$, which maps all elements other than $\overline{obj_n}$ to a special value *failure*. We use *hp* to range over all heap states.

A *frame* is created each time a method is invoked, which contains a *local variable table* and an *operand stack* for the method. A frame is destroyed when the method completes.

A *local variable table state* is a list of the form $[wrd_0, \cdots, wrd_n]$ with $n \geq -1$. We use *lvs* to range over all local variable table states. Each method has a fixed number of local variables.

An *operand stack state* is a list $[wrd_0, \cdots, wrd_n]$ with $n \geq -1$. We use *stk* to range over all operand stack states. Each method has a fixed maximal length of operand stacks.

Note that we need not define formally what a frame is, since no frames are explicitly used in our specification.

Each JVM thread has a *Java stack* to store at least the old current frame and a return address upon a method invocation. When the method invocation completes normally, the old current frame becomes the current frame and the return address becomes the current program point. In this chapter the Java stack contains tuples (lvs, stk, pp), where *lvs* is the old current local variable table state, *stk* the old current operand stack state and *pp* a return address. Since we do not consider multi-threads, we need only to consider one Java stack. We use *jstk* to range over all Java stack states.

A *program state* is a tuple of the form $(pp, jstk, lvs, stk, hp)$. We use *stat* to range over all program states.

4 Static types

Figure 1 defines all *static types*. In the static analysis, a memory location at a program point may obtain a static type, indicating the types of the runtime data that the memory location may hold at that program point in all executions. For simplicity, we may omit the phrases "at a program point" and "in all executions" in the rest of this chapter.

Reference type set	$\{\overline{ref_n}\}$ $(n > 0)$ where each ref_i is either type *null*, or a class or interface name as in `Java`.
Primitive type	*int*
Subroutine type	$sbr(pp)$ | *invldsbr*
Raw object type	$unin(pp, cnam)$ | $init(cnam)$
Unusable value type	*unusable*

Figure 1: Static types

We introduce the static type *null*. If a memory location may hold nothing more than the special object reference *null*, then the memory location may be given the static type *null*.

The static type *null* and class or interface names in `Java` are called *reference types*. Note that `java.lang.Object` (short: *Object*) is a class name in `Java`.

A nonempty *reference type set* is a static type. Intuitively, if a memory location may hold nothing more than *null* and objects that are of the reference types ref_i for $i = 1, \cdots, n$ but not raw objects (see below), then it may obtain the static type $\{\overline{ref_n}\}$.

It is worth mentioning that the Sun's implementation does not implement the concept of reference type sets in the bytecode verifier.

In our specification, a single reference type is always regarded as identical to the singleton set containing the reference type.

If a memory location may hold nothing more than elements of the primitive type *int*, then it may obtain the static type *int*.

As mentioned before, the byte offset of a `jsr` instruction can be regarded as an element of the subroutine type corresponding to the called subroutine. If a memory location may hold nothing more than some valid byte offsets of `jsr` instructions that call one common subroutine starting at *pp*, then the memory location may obtain a *subroutine type* $sbr(pp)$ as its static type. Note that $sbr(pp) \neq sbr(pp')$ if and only if $pp \neq pp'$.

If a memory location may hold some valid and invalid byte offsets of `jsr` instructions, then the memory location may obtain the static type *invldsbr*.

The forms $unin(pp, cnam)$ and $init(cnam)$ are static types for memory locations holding *raw objects*. More concretely, if a memory location may hold nothing more than objects of the class *cnam* created by one common `new` instruction at a program point *pp*, then the memory location may obtain the static type $unin(pp, cnam)$. If the memory location may hold nothing more than an object that is being currently initialized by an instance initialization method for the class *cnam* and has not encountered another instance initialization method within the current instance initialization method, then the memory location may obtain the static type $init(cnam)$. Note that

$unin(pp, cnam) \neq unin(pp', cnam')$ if and only if $pp \neq pp'$ or $cnam \neq cnam'$, $init(cnam) \neq init(cnam')$ if and only if $cnam \neq cnam'$.

Any memory location may obtain the static type *unusable*. In particular, if a memory location may hold runtime data of incompatible types, then it should obtain the static type *unusable*, indicating that the content of the memory location is unusable in practice. For example, if a local variable may hold an object and an element of the type *int*, then our specification will enforce the local variable to obtain the static type *unusable*.

To represent the above intuitive semantics more precisely, we define a partial order $\sqsupseteq$ on static types as the smallest reflexive and transitive relation satisfying that

$$
\begin{array}{ll}
\{\overline{ref_n}\} \sqsupseteq \{\overline{ref_m}\} & \text{for all } n \text{ and } m \text{ with } n \leq m \text{ and all } ref_i, i = 1, \ldots, m \\
invldsbr \sqsupseteq sbr(pp) & \text{for all } pp \\
unusable \sqsupseteq any & \text{for all static types } any
\end{array}
$$

The relation $any \sqsupseteq any'$ is read as "*any covers any'*".

Intuitively, if *any* covers *any'*, then any instruction applicable to a memory location with *any* is also applicable to a memory location with *any'*. Note that the relation implies that, for example, if *any* covers both *int* and a reference type *ref*, then *any* must be *unusable*.

5 Short notations for zero or one of several static types

The syntax in Figure 2 means that an identifier on the left of ::= denotes an arbitrary static type or the identifier *void* that either explicitly occurs or is denoted by an identifier on the right of ::=.

Conceptually, the identifier *void* is not a static type. It is just an auxiliary identifier denoting the situation that no static type is present.

For example, *ref* denotes an arbitrary class or interface name or *null*, *tys* denotes an arbitrary reference type set or a primitive type, *notnull_void* denotes an arbitrary class or interface name or *void*, and *any* denotes an arbitrary static type.

6 Program point types and program types

In general, there is no guarantee that any `class` file that is asked to be loaded is properly formed. Thus according to the OJVMS, the bytecode verifier should ensure that the `class` file satisfies some constraints. In particular, the bytecode verifier should be able to statically derive a static type for each local variable and operand stack entry at each program point, and ensure that the derived static types satisfy some constraints.

For this purpose, we define a *local variable table type* as a list of the form $[any_0, \cdots, any_n]$ with $n \geq -1$. We use *lvsty* to range over all local variable

Class name	*cnam*	::=	an arbitrary class name
Interface name	*inam*	::=	an arbitrary interface name
Reference type	*ref*	::=	*cnam* | *inam* | *null*
Primitive type	*prim*	::=	*int*
Void type	*void*	::=	
Type that is not *null*	*notnull*	::=	*cnam* | *inam* | *prim*
Type	*ty*	::=	*ref* | *prim*
Reference type set	*refs*	::=	$\{\overline{ref_n}\}$ $(n > 0)$
Type set	*tys*	::=	*refs* | *prim*
Subroutine type	*sbr*	::=	*sbr*(*pp*) | *invldsbr*
Raw object type	*raw*	::=	*unin*(*pp*, *cnam*) | *init*(*cnam*)
Type or void	*notnull_void*	::=	*notnull* | *void*
Reference type set or raw object type	*refs_raw*	::=	*refs* | *raw*
Reference type set, raw object type or subroutine type	*refs_raw_sbr*	::=	*refs* | *raw* | *sbr*
Anything	*any*	::=	*tys* | *raw* | *sbr* | *unusable*

Figure 2: Auxiliary symbols denoting zero or one of several static types

table types. For $lvsty = [any_0, \cdots, any_n]$ and $lvsty' = [any_0, \cdots, any_m]$, we define that $lvsty \sqsupseteq lvsty'$ holds if and only if $n = m$ and $any_i \sqsupseteq any'_i$ hold for all $i = 0, \ldots, n$.

We define an *operand stack type* as a list of the form $[any_0, \cdots, any_n]$ with $n \geq -1$. We use *stkty* to range over all operand stack types. For $stkty = [any_0, \cdots, any_n]$ and $stkty' = [any_0, \cdots, any_m]$, we define that $stkty \sqsupseteq stkty'$ holds if and only if $n = m$ and $any_i \sqsupseteq any'_i$ hold for all $i = 0, \ldots, n$.

The above definitions that a local variable or an operand stack entry can hold values of arbitrary static types.

To record whether an instance initialization method has been called inside another instance initialization method, we use three *initialization tags*, namely *needIn*, *needNoIn* and *unknown*. We use *intag* to range over all initialization tags. A $\sqsupseteq$-relation is defined on these tags as follows:

$$intag \sqsupseteq intag' \text{ if and only if } intag = unknown \text{ or } intag = intag'$$

We define a *program point type* as a tuple $(lvsty, stkty, intag, mod)$ where *mod* will be defined in Section 10.6. We use *ptty* to range over all program point types.

Let $ptty = (lvsty, stkty, intag, mod)$ and $ptty' = (lvsty', stkty', intag', mod')$. The relation $ptty \sqsupseteq ptty'$ holds if and only if $lvsty \sqsupseteq lvsty'$, $stkty \sqsupseteq stkty'$, $intag \sqsupseteq intag'$ and $mod \sqsupseteq mod'$ hold, where the last relation will be defined in Section 10.6.

Intuitively, the relation $ptty \sqsupseteq ptty'$ is used to ensure that any instruction that is applicable to all program states of the program point type *ptty* must be applicable to all program states of the program point type $ptty'$.

For the program *Prg*, a *program type* is a mapping $\{pp \mapsto ptty_{pp} \mid pp \in allPP(Prg)\}$. We use *prgty* to range over all program types. Let *prgty* and $prgty'$ be two program types. Then we define that $prgty \sqsupseteq prgty'$ holds if and only if $prgty(pp) \sqsupseteq prgty'(pp)$ holds for all $pp \in allPP(Prg)$. These concepts can also be defined for the fixed method *Mth*.

7 The reference type hierarchy

A reference type hierarchy in the JVM is as in `Java`. Following the OJVMS (§ 2.6.4), we formally define a subtyping relation *widRefConvert* as the smallest reflexive transitive relation on all reference type sets *refs* satisfying:

$$
\begin{array}{rl}
widRefConvert(cnam, cnam') & \text{if } cnam \ \texttt{extends} \ cnam' \\
widRefConvert(cnam, inam) & \text{if } cnam \ \texttt{implements} \ inam \\
widRefConvert(inam, inam') & \text{if } inam \ \texttt{extends} \ inam' \\
widRefConvert(inam, Object) & \\
widRefConvert(null, ref) & \\
widRefConvert(\{\overline{ref_n}\}, \{\overline{ref'_m}\}) & \text{if } \forall(1 \le i \le n).\exists(1 \le j \le m). \\
 & \quad widRefConvert(ref_i, ref'_j)
\end{array}
$$

Note that we do not consider array types. We use the relation *diSubcls* to denote the direct subclass relation on classes.

To constrain the types of the actual and formal parameters in a method invocation we define the relation *invoConvert* on all reference type sets and the primitive type *int* as

$$invoConvert := widRefConvert \cup \{(int, int)\}$$

Note that $\{(int, int)\}$ is a degenerate case of the widening primitive conversion in the OJVMS (§ 2.6.2). It suffices for us to have the degenerate case, since we consider only one primitive type *int*.

To constrain the types of the variable and the value in an assignment, we define the relation *assConvert* on all reference type sets and the primitive type *int* as

$$assConvert := invoConvert.$$

The OJVMS requires that *assConvert* extends *invoConvert* by some narrowing primitive conversions for integer constants. We do not consider this difference for simplicity.

Intuitively, if a reference type set contains both a super- and a subtype, then the subtype is redundant. Practically, a Bytecode Verifier could implement elimination of redundant reference types from a reference type set with respect to a subtyping relation as an optimization step.

8 Constant pool resolution

According to the OJVMS, each class (or interface) should have a *constant pool* whose entries name entities like classes, interfaces, methods and fields referenced from the code of the class (or interface, respectively) or from other constant pool entries. An individual instruction in the class (or interface, respectively) may carry an index of an entry in the constant pool, and during the execution of this instruction, the JVM is responsible for resolving the entry, i.e. determining a concrete entity from the entry. This process of resolving an entry is called *constant pool resolution*.

For our formal specification we introduce some defined functions, called *resolution functions*, which hide the details of resolution. In fact, except that the resolution processes should take correct sorts of data as argument and yield correct sorts of data or a failure as result, other details are not important at all for the formal specification and proofs in this chapter. Nevertheless, we still explain the definitions of the resolution functions, in order to give a feeling why the resolution functions here are proper abstractions of the real resolution processes.

A resolution function in this section often takes as parameter two one-byte-wide integers $ind1$ and $ind2$, which build the index $\mathit{offset}(ind1, ind2)$ in a constant pool. In this sense, a resolution function has always a constant pool as an implicit parameter.

The resolution function $\mathit{cResol}(index1, index2)$ yields a class name $cnam$.

For any $cnam$, we define a function

$$\mathit{allFields}(cnam) := \{(\mathit{fnam}, \mathit{notnull}) \mid \mathit{fnam} \text{ and } \mathit{notnull} \text{ are the name and type of a field in the class } cnam\}$$

Note that a field in the class $cnam$ is either directly defined in the class or in a superclass of the class. Since a field name fnam is a qualified name, we need not consider the problem with hiding of fields.

We define a resolution function for a field as

$$\mathit{fResol}(ind1, ind2) := (\mathit{fnam}, cnam, \mathit{notnull})$$

where fnam is the name of the field, $cnam$ the class containing the field declaration, and $\mathit{notnull}$ the type of the field.

We define a resolution function for a special method as

$$\mathit{mResolSp}(ind1, ind2) := (mnam, cnam, (\overline{ty_n})\mathit{notnull\_void}, cod, mxl)$$

where $mnam$ is the name of the method, $cnam$ the class containing the declaration of the method, $(\overline{ty_n})\mathit{notnull\_void}$ the descriptor of the method, cod the method code and mxl the length of the local variable table in the method.

We define a resolution function for a `static` method as

$$\mathit{mResolSt}(ind1, ind2) := (mnam, cnam, (\overline{ty_n})\mathit{notnull\_void}, cod, mxl).$$

We define a resolution function for an instance method[1] as

$$mResolV(ind1, ind2) := (mnam, cnam, (\overline{ty_n})notnull\_void).$$

But the function $mResolV(ind1, ind2)$ does not yield a method code. For doing this, we need to define another function

$$mSelV(obj, mnam, (\overline{ty_n})notnull\_void) := (cod, mxl)$$

which takes an object *obj*, and yields the method code *cod* for the object *obj* and the length *mxl* of the local variable table in the method.

We define a resolution function for an interface method as

$$mResolI(ind1, ind2) := (mnam, inam, (\overline{ty_n})notnull\_void)$$

where *inam* is the name of the interface, instead of the class, that contains the declaration of the method. Furthermore, we define

$$mSelI(obj, mnam, (\overline{ty_n})notnull\_void) := (cod, mxl).$$

For convenience we define the auxiliary function

$$mInfo(pp) := (mnam, cnam, (\overline{ty_n})notnull\_void, mxl)$$

where *mnam* is the method containing the *pp*, $(\overline{ty_n})notnull\_void$ the descriptor of the method, *cnam* the class containing the declaration of the method, and *mxl* the number of the local variables in the method.

9 Constraint domain and constraints

The previous sections have in fact introduced (part of) a constraint domain for our formal specification. Although there are no problems to completely formally define all concepts in a constraint domain, we can only discuss (part of) them informally in this chapter due to the space limit.

First of all, all data, data structures (e.g. local variable table states, operand stack states, program states), static types and type structures (e.g. local variable table types, operand stack types, program point types, pogram types) defined in the previous sections are elements of the constraint domain. These elements are all *sorted*. Informally, every time when we introduce an identifier to range over a kind of data, data structures, static types or type structures, we introduce a sort. We use the introduced identifiers also as names of these sorts. So it is possible for one sort to have several names. For example, the sort *byt* consists of all one-byte wide data, the sort *wrd* all one-word wide data, the sort *pp* all program points, the sort *lvs* all local variable

[1] Thanks to Gilad Bracha for clarifying comments on the semantics of method dispatch at this point.

table states, the sort *stk* all operand stack states, the sort *stat* all program states, the sorts *ref*, *refs*, *tys* and $refs\_raw$ corresponding static types, respectively, as defined in Figure 2, the sort *lvsty* all local variable table types, the sort *stkty* all operand stack types and the sort *prgty* all program point types. Standard data or type structures, e.g. sets or lists of some data or types, also build sorts, but not necessarily have a sort name.

There is a subsort relation among the sorts, which corresponds to the subset relation. In particular, Figure 2 defines that if a sort occurs as an alternative on the right of ::=, then the sort on the left of ::= is a supersort of it. For example, the sort *ref* is a supersort of the sorts *cnam* and *inam* and contains *null*, the sort *prim* contains *int*, the sort *notnull* is a supersort of the sorts *cnam*, *inam* and *prim*, *refs* contains all $\{\overline{ref_n}\}$, where each ref_i is an element of the sort *ref*, etc. Since a singleton reference type set $\{ref\}$ and the reference type *ref* are regarded as the same static type, we define that the sort *refs* is a supersort of the sort *ref*.

For each sort, there are a countable set of variables. In general, the completely capitalized version of a sort name denotes a variable of the sort. For example, BYT is a variable of *byt* and WRD a variable of *wrd*. For notional simplicity we also introduce the variable P for the sort *pp*, L for the sort *lvs*, S for the sort *stk*, J for the sort *jstk*, H for the sort *hp*, LG for the sort *lvstag*, SG for the sort *stktag*, Ξ for the sort *stat*, LT for the sort *lvsty*, ST for the sort *stkty*, IT for the sort *intag*, M for *mod*, Π for the sort *ptty* and Φ for the sort *prgty*. We use _ to denote a wildcard variable.

In general, terms are built using variables, constants and functions in the constraint domain. Terms are sorted as usual. A sort of a subsort is always a term of a supersort. Every term has a least sort. We will use the partially capitalized version of a sort name, where only the first letter is changed into a capital letter, to range over all terms of the sort. For example, *Pp*, *Stat* and *Ptty* range over the terms in the sorts *pp*, *stat* and *ptty*, respectively.

A term containing no variables is called *closed*. In fact, each element in the constraint domain is a closed term.

Logical formulas are built as in First-Order Predicate Logic, where predicates take only sorted arguments in the constraint domain. We use q and r to range over all logical formulas.

We use the form $q[\overline{s_n}]$ to denote a logical formula containing the (occurrences of) terms $\overline{s_n}$. If the forms $q[\overline{s_n}]$ and $q[\overline{t_n}]$ occur in the same context (e.g. the same rule), then s_i and t_i are of the same sort for $i = 1, \ldots, n$, and $q[\overline{t_n}]$ is the logical formula obtained from $q[\overline{s_n}]$ by replacing each s_i by t_i for $i = 1, \ldots, n$.

A *substitution* is a finite mapping of the form $\{\overline{X_n \mapsto s_n}\}$, where the sort of each term s_i must be a subsort of the sort of X_i for all $i = 1, \cdots, n$. We consider only *closed* substitutions in this chapter, i.e. where s_i is a closed term for $i = 1, \ldots, n$. We use σ to range over all closed substitutions.

A *constraint* is a logical formula. A set of constraints $\{q_1, \cdots, q_m\}$ represents the logical formula $q_1 \wedge \cdots \wedge q_m \wedge true$.

A constraint q is *satisfied under* a substitution σ if and only if $\sigma(q)$ is closed and holds in the constraint domain. A constraint q is *satisfiable* if there is a substitution, under which the constraint q is satisfied.

In our formal specification, we may define a function f that yields results in a sort a for some arguments and the special value *failure* not in a for all other arguments, and use a term $f(\overline{s_n})$ in a constraint, say $q[f(\overline{s_n})]$, where a term of a is required. Intuitively, this usage always implicitly requires that $f(\overline{s_n})$ should not yield *failure*. Formally, we may always define a new sort a', which is a supersort of a and contains the *failure* as a constant, define the f to have the result sort a', and replace the constraint $q[f(\overline{s_n})]$ by the constraints $q[X]$ and $X = f(\overline{s_n})$, where X is a new variable of the sort a. The reason why the constraint $X = f(\overline{s_n})$ assures that "$f(\overline{s_n})$ is not equal to *failure*" is that *failure* is not in the sort a and thus $X = \mathit{failure}$ is never satisfiable. (Note that if there are two functions yielding *failure*, then we need to assume that they yield different *failure*'s; otherwise the least sort of the term *failure* may not exist.)

Our formal specification consists of two parts. The first part defines a state transition relation on program states $\mathit{stat} \implies \mathit{stat}'$, read as "*stat* changes into *stat'*". The relation is defined by state transition rules of the following form:

$$\frac{\mathit{Premises}}{\Xi[\overline{s_n}] \implies \Xi[\overline{t_n}]}$$

where *Premises* is a set of constraints. Let

$$Q := \mathcal{FV}(\mathit{Premises}) \cup \mathcal{FV}(\Xi[\overline{s_n}] \implies \Xi[\overline{t_n}]).$$

Then the rule means that if all constraints in *Premises* are satisfied under a substitution σ, then $\sigma(\Xi)[\overline{\sigma(s_n)}] \implies \sigma(\Xi)[\overline{\sigma(t_n)}]$ holds. In the sequel, we may also say that $\Xi[\overline{s_n}]$ changes into $\Xi[\overline{t_n}]$ in the informal discussion for simplicity.

To specify all program types of a program, the following two forms of constraints are particularly important:

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

The former says that the program point type at Pp in *Prgty* is *Ptty*. The latter says that the program point type at Pp in *Prgty* covers *Ptty*. If a program point Pp can be reached by more than one preceding program point, then it is quite convenient to write a constraint of the latter form to constrain the program point type at Pp.

The type system in our formal specification should introduce constraints on one program type for the method *Mth*. Therefore, we require that all typing rules contain one common program type variable Φ.

In general, a typing rule is in the form:

$$\frac{\begin{array}{c}\boxed{ACconds}\\ CConstrs\end{array}}{SConstrs}$$

The *ACconds* is a set of logical formulas, called *applicability conditions*, and contains a distinguished constraint $Mth(P) = Instr$. The term *Instr* gives the form of an instruction. Intuitively, *ACconds* is used to determine a program point P, where the rule can be applied. The identifier *CConstrs* stands for a set of logical formulas. It contains no logical formulas of the form $\Phi(P) \sqsupseteq Ptty$. Intuitively, *CConstrs* constrains $\Phi(P)$. The identifier *SConstrs* stands for a set of logical formulas of the form $\Phi(Pp') \sqsupseteq Ptty$, where in most cases Pp' stands for a successor program point.

The reason for us to write a typing rule in the form as above is that a typing rule also suggests an intuitive data-flow analysis step. Roughly speaking, if the data-flow analysis arrives at a program point Pp satisfying *ACconds*, in particular the constraint $Mth(Pp) = Instr$ in *ACconds*, and if the program type at Pp satisfies *CConstrs*, then the program type at each successor program point Pp' should satisfy the corresponding constraint in *SConstrs*.

Let

$$\begin{array}{l} Q := \mathcal{FV}(ACconds) - \{\Phi\} \\ Q' := \mathcal{FV}(CConstrs \cup SConstrs) - (\{\Phi\} \cup Q) \end{array}$$

then a typing rule as above formally introduces the constraint

$$\forall Q.(ACconds \Rightarrow \exists Q'.(CConstrs \cup SConstrs))$$

It is easy to see that the constraint holds if and only if, if *ACconds* is satisfied under a substitution σ with $\mathcal{D}om(\sigma) = Q \cup \{\Phi\}$, then there is a substitution σ' with $\mathcal{D}om(\sigma) = Q \cup Q' \cup \{\Phi\}$ such that $\sigma'_{|Q \cup \{\Phi\}} = \sigma$ and $\sigma'(CConstrs \cup SConstrs)$ hold.

Let $Constraints_{Mth}$ denote the set of the constraints introduced as above from all typing rules. Then we say that the method *Mth has a program type prgty*, or that a program type *prgty is a program type* of the method *Mth*, if and only if all constraints in $Constraints_{Mth}$ are satisfied under $\{\Phi \mapsto prgty\}$. Note that a program may have more than one program type. For example, a local variable that is not used in a method may be given an arbitrary static type in a program type. A program is said to be *statically well-typed* if and only if it has a program type.

10 The rules in the formal specification

There are constraints that should occur in many rules. We omit the explicit presentation of the following constraints for notational simplicity.

- The *CConstrs* in a typing rule always implicitly contains a constraint $Pp \in allPP(Mth)$ for each $\Phi(Pp) \sqsupseteq Ptty$ in the *SConstrs*. This assures that Pp is always a program point, i.e. a starting address of an instruction.

– In the specification we only consider the instructions for one-word wide data. Thus the rules are all based on the assumption that all data in local variables and the operand stack are one-word wide.

10.1 Load and store instructions

The state transitions for loading and storing objects and integers of type *int* are defined by the rules in Figure 3. The `aload` and `iload` instructions load a local variable onto the operand stack. The `astore` and `istore` instructions store a value from the operand stack in a local variable.

$$\frac{Prg(P) = \texttt{aload}\ IND \text{ or } \texttt{iload}\ IND}{\varXi[P, L, S] \implies \varXi[P+2, L, S + L(IND)]} \qquad \text{(S-1),(S-2)}$$

$$\frac{Prg(P) = \texttt{astore}\ IND \text{ or } \texttt{istore}\ IND}{\varXi[P, L, S + WRD] \implies \varXi[P+2, L[IND \mapsto WRD], S]} \qquad \text{(S-3),(S-4)}$$

Figure 3: The state transitions for load and store instructions

The typing rules for load and store instructions are given in Figure 4. We explain rule (T-1) to show some of the tricky points in the formulation of constraints. First, $REFS\_RAW = LT(IND)$ expresses a membership constraint, i.e. that the static type $LT(IND)$ should be in the sort *refs_raw*, since $REFS\_RAW$ can only be instantiated by an element in the sort *refs_raw*. It implies that an `aload` instruction can load both initialized and uninitialized objects. In addition, rule (T-1) says that the local variable table type at $P+2$ (i.e. after the instruction) should componentwise cover that at P (before the instruction). The same should also hold for the operand stack type, except that the operand stack type at $P+2$ should be extended by the static type of the IND-th local variable. A similar constraint should also hold on the components M at P and Mod' at $P+2$. The precise definitions of M and Mod' will be given in Sections 10.6 and 10.7. Note that the variables $\varPhi$ and LT in the terms $\varPhi(P)$ and $LT(IND)$ are not higher-order (i.e. function) variables, since the terms of this form in this chapter can always be regarded as applications of an implicit function *app* on two first-order arguments.

Similar explanations can be given for the other three typing rules. One point that is worth noticing in rule (T-3) is that the variable $REFS\_RAW\_SBR$ can be initiantiated into an element of the sort *sbr*, whereas the variable $REFS\_RAW$ in rule (T-1) cannot. This means that, as required in the OJVMS and implemented in the Sun's implementation, an `astore` instruction can store a (valid or invalid) byte offset, whereas an `aload` instruction cannot load it.

An `aconst_null` instruction loads the reference *null*. Its state transition rule and typing rule are defined in Figure 5.

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{aload}\ IND}\\ \Phi(P) = \Pi[LT, ST, M]\\ REFS\_RAW = LT(IND)\end{array}}{\Phi(P+2) \sqsupseteq \Pi[LT, ST + REFS\_RAW, Mod']} \tag{T-1}$$

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{iload}\ IND}\\ \Phi(P) = \Pi[LT, ST, M]\\ int = LT(IND)\end{array}}{\Phi(P+2) \sqsupseteq \Pi[LT, ST + int, Mod']} \tag{T-2}$$

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{astore}\ IND}\\ \Phi(P) = \Pi[LT, ST + REFS\_RAW\_SBR, M]\end{array}}{\Phi(P+2) \sqsupseteq \Pi[LT[IND \mapsto REFS\_RAW\_SBR], ST, Mod']} \tag{T-3}$$

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{istore}\ IND}\\ \Phi(P) = \Pi[LT, ST + int, M]\end{array}}{\Phi(P+2) \sqsupseteq \Pi[LT[IND \mapsto int], ST, Mod']} \tag{T-4}$$

Figure 4: The typing rules for load and store instructions

$$\frac{Prg(P) = \texttt{aconst\_null}}{\Xi[P, S] \implies \Xi[P+1, S + null]} \tag{S-5}$$

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{aconst\_null}}\\ \Phi(P) = \Pi[ST]\end{array}}{\Phi(P+1) \sqsupseteq \Pi[ST + null]} \tag{T-5}$$

Figure 5: The state transitions for `aconst_null` and `bipush`

The state transitions for `getfield` and `putfield` are defined in Figure 6. A `getfield` instruction replaces an object reference at the top of the operand stack by the content of a field of the referenced object.

A `putfield` instruction stores the content at the top of the operand stack into a field of the object referenced by the second top of the operand stack.

$$\frac{\begin{array}{l} \mathit{Prg}(P) = \texttt{getfield}\ \mathit{IND}1\ \mathit{IND}2 \\ (\mathit{FNAM}, \_, \mathit{NOTNULL}) = \mathit{fResol}(\mathit{IND}1, \mathit{IND}2) \\ \mathit{WRD} = H(\mathit{OBJ})(\mathit{FNAM}) \end{array}}{\Xi[P, S + \mathit{OBJ}, H] \implies \Xi[P+3, S + \mathit{WRD}, H]} \tag{S-6}$$

$$\frac{\begin{array}{l} \mathit{Prg}(P) = \texttt{putfield}\ \mathit{IND}1\ \mathit{IND}2 \\ (\mathit{FNAM}, \_, \_) = \mathit{fResol}(\mathit{IND}1, \mathit{IND}2) \\ \mathit{REC} = H(\mathit{OBJ})[\mathit{FNAM} \mapsto \mathit{WRD}] \end{array}}{\Xi[P, S + \mathit{OBJ} + \mathit{WRD}, H] \implies \Xi[P+3, S, H[\mathit{OBJ} \mapsto \mathit{REC}]]} \tag{S-7}$$

Figure 6: The state transitions for `getfield` and `putfield`

The typing rules for `getfield` and `putfield` are given in Figure 7. The sort of the variable *REFS* in $\Pi[ST + REFS]$ and $\Pi[ST + REFS + TYS]$ assures that the *OBJ* in Figure 6 really references an object. The constraint $\mathit{widRefConvert}(\mathit{REFS}, \mathit{CNAM})$ assures that in Figure 6 if $\mathit{OBJ} \in \mathcal{D}om(H)$, then $\mathit{FNAM} \in \mathcal{D}om(H(\mathit{OBJ}))$ holds, i.e. both $H(\mathit{OBJ})(\mathit{FNAM})$ and $H(\mathit{OBJ})[\mathit{FNAM} \mapsto \mathit{WRD}]$ are defined and make sense. But the typing rules do not ensure that the condition $\mathit{OBJ} \in \mathcal{D}om(H)$ in Figure 6 holds, since the *OBJ* may hold *null* at run time. If $\mathit{OBJ} \notin \mathcal{D}om(H)$ holds, then $H(\mathit{OBJ})$ yields a *failure*. Thus the *Premises* in both rules in Figure 6 are not satisfiable. In fact, in this case we would need another state transition rule to describe which kind of runtime exception can be thrown. However, as mentioned before, our formal specification does not consider this.

10.2 Object creation

A `new` instruction creates an object. The state transition and typing rules for the instruction are defined in Figure 8.

The condition $\mathit{OBJ} \notin \mathcal{D}om(H)$ in rule (S-8) assures that the object reference *OBJ* is new. Rule (T-8) says that the operand stack type after the instruction covers one with $\mathit{unin}(P, \mathit{CNAM})$[2] at the top, which indicates that the operand stack may hold an object that has not been initialized by an instance initialization method `<init>`, i.e. an uninitialized object. Indeed, a

[2] The OJVMS mentions such a type but gives no details on how it can be used in the specification.

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{getfield}\ IND1\ IND2} \\ \Phi(P) = \Pi[ST + REFS] \\ (\_, CNAM, NOTNULL) = fResol(IND1, IND2) \\ widRefConvert(REFS, CNAM)\end{array}}{\Phi(P+3) \sqsupseteq \Pi[ST + NOTNULL]} \quad \text{(T-6)}$$

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{putfield}\ IND1\ IND2} \\ \Phi(P) = \Pi[ST + REFS + TYS] \\ (\_, CNAM, NOTNULL) = fResol(IND1, IND2) \\ widRefConvert(REFS, CNAM) \\ assConvert(TYS, NOTNULL)\end{array}}{\Phi(P+3) \sqsupseteq \Pi[ST]} \quad \text{(T-7)}$$

Figure 7: The typing rules for `getfield` and `putfield`

$$\frac{\begin{array}{l}Prg(P) = \texttt{new}\ IND1\ IND2 \\ CNAM = cResol(IND1, IND2) \\ OBJ \notin \mathcal{D}om(H) \\ \mathcal{D}om(REC) = allFields(CNAM)\end{array}}{\Xi[P, S, H] \implies \Xi[P+3, S + OBJ, H[OBJ \mapsto REC]]} \quad \text{(S-8)}$$

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{new}\ IND1\ IND2} \\ \Phi(P) = \Pi[LT, ST] \\ CNAM = cResol(IND1, IND2) \\ unin(P, CNAM) \notin LT \\ unin(P, CNAM) \notin ST\end{array}}{\Phi(P+3) \sqsupseteq \Pi[LT, ST + unin(P, CNAM)]} \quad \text{(T-8)}$$

Figure 8: The state transition and the typing rule for `new`

typing rule that forbids the use of a memory location with a static type of the form $unin(\_,\_)$ forbids the use of an uninitialized object.

The constraints $unin(P, CNAM) \notin LT$ and $unin(P, CNAM) \notin ST$ assure that at the program point P, no new object created by the same `new` instruction at P can be used as a new object. This is strictly weaker than to say that there is no new object created by the `new` instruction at P, since a memory location at P is still allowed to hold a new object created by the `new` instruction at P if the memory location has the type *unusable*. For an example, see Section 11.

10.3 Operand stack management instructions

We only give the rules for `dup` in Figure 9. The rules for other instructions are similar.

$$\frac{Prg(P) = \mathtt{dup}}{\Xi[P, S + WRD] \implies \Xi[P+1, S + WRD + WRD]} \tag{S-9}$$

$$\frac{\boxed{Mth(P) = \mathtt{dup}} \quad \Phi(P) = \Pi[ST + ANY]}{\Phi(P+1) \sqsupseteq \Pi[ST + ANY + ANY]} \tag{T-9}$$

Figure 9: The state transition and typing rules for `dup`

10.4 Control transfer instructions

$$\frac{\begin{array}{l} Prg(P) = \mathtt{if\_acmpeq}\ BYT1\ BYT2 \\ OBJ1 = OBJ2 \Rightarrow P' = P + \mathit{offset}(BYT1, BYT2) \\ OBJ1 \neq OBJ2 \Rightarrow P' = P + 3 \end{array}}{\Xi[P, S + OBJ1 + OBJ2] \implies \Xi[P', S]} \tag{S-10}$$

$$\frac{\begin{array}{l} Prg(P) = \mathtt{if\_icmpeq}\ BYT1\ BYT2 \\ VAL1 = VAL2 \Rightarrow P' = P + \mathit{offset}(BYT1, BYT2) \\ VAL1 \neq VAL2 \Rightarrow P' = P + 3 \end{array}}{\Xi[P, S + VAL1 + VAL2] \implies \Xi[, S]} \tag{S-11}$$

$$\frac{Prg(P) = \mathtt{goto}\ BYT1\ BYT2}{\Xi[P] \implies \Xi[\mathit{offset}(BYT1, BYT2)]} \tag{S-12}$$

Figure 10: The state transitions for control transfer instructions

$$\frac{\boxed{Mth(P) = \texttt{if\_acmpeq}\ BYT1\ BYT2} \quad \Phi(P) = \Pi[ST + REFS + REFS']}{\Phi(P + \mathit{offset}(BYT1, BYT2)) \sqsupseteq \Pi[ST] \quad \Phi(P+3) \sqsupseteq \Pi[ST]} \tag{T-10}$$

$$\frac{\boxed{Mth(P) = \texttt{if\_icmpeq}\ BYT1\ BYT2} \quad \Phi(P) = \Pi[ST + int + int]}{\Phi(P + \mathit{offset}(BYT1, BYT2)) \sqsupseteq \Pi[ST] \quad \Phi(P+3) \sqsupseteq \Pi[ST]} \tag{T-11}$$

$$\frac{\boxed{Mth(P) = \texttt{goto}\ BYT1\ BYT2} \quad \Phi(P) = \Pi}{\Phi(P + \mathit{offset}(BYT1, BYT2)) \sqsupseteq \Pi} \tag{T-12}$$

Figure 11: The typing rules for control transfer instructions

All control transfer instructions can be dealt with in a very similar way. We consider only a few control transfer instructions. The state transitions for these instructions are given in Figure 10. They are quite straightforward.

The OJVMS requires (page 133) that no uninitialized objects may exist on the operand stack or in a local variable when a control transfer instruction causes a backwards branch. In our specification this requirement is unnecessary, thanks to rule (T-8).

10.5 Method invocation and return instructions

The state transitions for method invocation instructions are defined in Figure 12. We first consider the state transition rule (S-13) for `invokespecial`. Since the instruction is only used to invoke instance instantiation methods `<init>` and private methods, and to perform method invocations via `super`, we use the function *mResolSp*. The state transition says that the execution of the invoked method starts with a program state, in which the operand stack is empty and the local variables hold the object, on which the method is invoked, and all actual arguments. We use the notation lvs^n to denote an arbitrary local variable table state with the length n.

The state transition for `invokevirtual` (or `invokeinterface`) is similar to that for `invokespecial`. The difference is only that the former uses the functions *mResolV* and *mSelV* (or *mResolI* and *mSelI*, respectively) to compute the method code associated with OBJ, whereas the latter uses the function *mResolSp* to do the same thing, independent of OBJ. Note that the bytes BYT and 0 in a `invokeinterface` instruction are useless. They are contained in the instruction for historical reasons.

Invocation of a method leads to the execution of a method code. The typing rule in Figure 13 constrains the program point type at the beginning

$$\frac{\begin{array}{l} Prg(P) = \texttt{invokespecial}\ IND1\ IND2 \\ (\_, \_, (\overline{TY_n})NOTNULL\_VOID, COD, MXL) = mResolSp(IND1, IND2) \end{array}}{\begin{array}{l} \Xi[P, J, L, S + OBJ\overline{+WRD_n}] \\ \Longrightarrow\ \Xi[COD, J + (L, S, P + 3), lvs^{MXL}[0 \mapsto OBJ, \overline{n \mapsto WRD_n}], []] \end{array}} \quad \text{(S-13)}$$

$$\frac{\begin{array}{l} Prg(P) = \texttt{invokevirtual}\ IND1\ IND2 \\ (MNAM, \_, (\overline{TY_n})NOTNULL\_VOID) = mResolV(IND1, IND2) \\ MNAM \neq <\texttt{init}> \\ (COD, MXL) = mSelV(OBJ, MNAM, (\overline{TY_n})NOTNULL\_VOID) \end{array}}{\begin{array}{l} \Xi[P, J, L, S + OBJ\overline{+WRD_n}] \\ \Longrightarrow\ \Xi[COD, J + (L, S, P + 3), lvs^{MXL}[0 \mapsto OBJ, \overline{n \mapsto WRD_n}], []] \end{array}} \quad \text{(S-14)}$$

$$\frac{\begin{array}{l} Prg(P) = \texttt{invokeinterface}\ IND1\ IND2\ BYT\ 0 \\ (MNAM, \_, (\overline{TY_n})NOTNULL\_VOID) = mResolI(IND1, IND2) \\ n = BYT - 1 \\ (COD, MXL) = mSelI(OBJ, MNAM, (\overline{TY_n})NOTNULL\_VOID) \end{array}}{\begin{array}{l} \Xi[P, J, L, S + OBJ\overline{+WRD_n}] \\ \Longrightarrow\ \Xi[COD, J + (L, S, P + 5), lvs^{MXL}[0 \mapsto OBJ, \overline{n \mapsto WRD_n}], []] \end{array}} \quad \text{(S-15)}$$

$$\frac{\begin{array}{l} Prg(P) = \texttt{invokestatic}\ IND1\ IND2 \\ (\_, \_, (\overline{TY_n})NOTNULL\_VOID, COD, MXL) = mResolSt(IND1, IND2) \end{array}}{\begin{array}{l} \Xi[P, J, L, S\overline{+WRD_n}] \\ \Longrightarrow\ \Xi[COD, J + (L, S, P{+}3), lvs^{MXL}[i \mapsto WRD_{i+1} \mid 0 \leq i < n], []] \end{array}} \quad \text{(S-16)}$$

Figure 12: The state transitions for method invocation instructions

$$\frac{\begin{array}{l} \boxed{\begin{array}{l} Mth(P) = \_ \\ P = (\_, 0) \end{array}} \\ (MNAM, CNAM, (\overline{TY_n})NOTNULL\_VOID, MXL) = mInfo(P) \\ MNAM = <\texttt{init}> \Rightarrow \\ \quad (\ NOTNULL\_VOID = void\ \wedge \\ \quad\quad (\ CNAM \neq Object \Rightarrow (\ LT = UU^{MXL}[0 \mapsto init(CNAM), \overline{n \mapsto TY_n}]\ \wedge \\ \quad\quad\quad\quad IT = needIn\ )\ )\ \wedge \\ \quad\quad (\ CNAM = Object \Rightarrow (\ LT = UU^{MXL}[0 \mapsto CNAM, \overline{n \mapsto TY_n}]\ \wedge \\ \quad\quad\quad\quad IT = needNoIn\ )\ ) \\ instMeth(MNAM, CNAM, (\overline{TY_n})NOTNULL\_VOID) \Rightarrow \\ \quad LT = UU^{MXL}[0 \mapsto CNAM, \overline{n \mapsto TY_n}] \\ statMeth(MNAM, CNAM, (\overline{TY_n})NOTNULL\_VOID) \Rightarrow \\ \quad LT = UU^{MXL}[i \mapsto TY_{i+1} \mid 0 \leq i < n] \end{array}}{\Phi(P) \sqsupseteq (LT, [], IT, mod_0)} \quad \text{(T-13)}$$

Figure 13: The typing rule for the starting program point of a method code

of a method code. The rule is totally independent of method invocation instructions. The rule says that the method must be a `<init>`, an instance or a `static` method. The static types given for the local variables depend on what kind method it is. In general, however, each local variable that does not store the object, on which the method is invoked, nor an actual parameter, is always given the type *unusable*. This means that the content of such a local variable cannot be used before the program explicitly assigns something to the local variable. This means that the content of such a local variable cannot be used before the program explicitly assigns something to the local variable. We use UU^n to denote the list $[unusable, \cdots, unusable]$ consisting of n *unusable*.

In the case of an `<init>` method, the local variable 0 stores the object being initialized. The static type for the local variable 0 and the initialization tag depend on whether the class $CNAM$ containing the method code is $Object$ or not. If $CNAM$ is not $Object$, then the initialization tag is *needIn* and the static type for the local variable 0 is $init(CNAM)$; The initialization tag *needIn* means that an instance initialization method needs to be called exactly once within the current method code in any case, since, as we will see, rule (T-14) will change the initialization tag into *needNoIn* and rule (T-20) checks whether the initialization tag is really *needNoIn*. Note that if $CNAM$ is $Object$, the object being initialized cannot, and need not, be initialized by another `<init>` within the current `<init>`.

Another point here is that the class $CNAM$ is chosen to be the one containing the `<init>` method. In fact, rule (T-14) will assure that $CNAM$ is either the original class of the object being initialized, or a superclass of it. Thus it is safe to use $CNAM$ at the place of the original class

The rule contains the component mod_0, which will be defined in Section 10.6.

The cases for an instance method and a `static` method are straightforward. Not much explanation for these rules is necessary.

The typing rules for method invocation instructions are given in Figure 14 and 15. Although these method invocation instructions are based on quite different mechanisms, they all require that the operand stack at the program point of the instruction contain the correct number of arguments with certain types. In order to express this, each of the typing rules contains constraints of the following forms:

$$(\cdots, (\overline{TY_n})void, \cdots) = a\_resolution\_function(IND1, IND2)$$
$$\Phi(P) = \Pi[\cdots, ST + \cdots \overline{+ TYS_n}, \cdots]$$
$$invoConvert(TYS_i, TY_i) \quad (i = 1, \ldots, n)$$

We consider rule (T-14) for `invokespecial` in Figure 14 in detail. The rule looks quite complicated, since the *CConstrs*-part of the rule basically gives three cases. The program point type at the program point of a `invokespecial` instruction must satisfy one of these cases.

$$
\frac{\begin{array}{l}
\boxed{Mth(P) = \texttt{invokespecial}\ IND1\ IND2} \\
(MNAM, CNAM, (\overline{TY_n})NOTNULL\_VOID, \_, \_) = mResolSp(IND1, IND2) \\
\Phi(P) = \Pi[LT, ST + REFS\_RAW\overline{+TYS_n}, IT, M] \\
invoConvert(TYS_i, TY_i) \quad (i = 1, \ldots, n) \\
MNAM = < \texttt{init} > \Rightarrow \\
\quad (\ (\ (\ REFS\_RAW = unin(P', CNAM)\ \wedge \\
\qquad\quad LT' = LT[CNAM/REFS\_RAW]\ \wedge \\
\qquad\quad ST' = ST[CNAM/REFS\_RAW]\ \wedge \\
\qquad\quad IT' = IT\ \wedge \\
\qquad\quad M' = Mod_1\ )\ \vee \\
\qquad (\ REFS\_RAW = init(CNAM')\ \wedge \\
\qquad\quad IT = needIn\ \wedge \\
\qquad\quad (CNAM' = CNAM \vee diSubcls(CNAM', CNAM))\ \wedge \\
\qquad\quad LT' = LT[CNAM/REFS\_RAW]\ \wedge \\
\qquad\quad ST' = ST[CNAM/REFS\_RAW]\ \wedge \\
\qquad\quad IT' = needNoIn\ \wedge \\
\qquad\quad M' = Mod_2\ )\ )\ \wedge \\
\qquad NOTNULL\_VOID = void\ ) \\
MNAM \neq < \texttt{init} > \Rightarrow \\
\quad (\ widRefConvert(REFS\_RAW, CNAM)\ \wedge \\
\qquad LT' = LT\ \wedge \\
\qquad (\ NOTNULL\_VOID = NOTNULL \Rightarrow ST' = ST + NOTNULL\ )\ \wedge \\
\qquad (\ NOTNULL\_VOID = void \Rightarrow ST' = ST\ )\ \wedge \\
\qquad IT' = IT\ \wedge \\
\qquad M' = M\ )
\end{array}}{\Phi(P+3) \sqsupseteq \Pi[LT', ST', IT', M']} \quad \text{(T-14)}
$$

Figure 14: The typing rule for `invokespecial`

The first case is when an `<init>` method is invoked on an object, on which no `<init>` method has been invoked before. In this case, the operand stack entry containing the object to be initialized has the static type $unin(P', CNAM)$. Following the OJVMS, the rule requires that the class containing the `<init>` method must be $CNAM$, and that after the instruction, all occurrences of $unin(P', CNAM)$ are changed into $CNAM$, indicating that the object has been initialized.

Note that the rule changes the component M into Mod_1 in the above case. The definition of Mod_1 will be given in Section 10.7.

The second case is when the instruction invokes an `<init>` method on an object that is being initialized within the enclosing `<init>` method, i.e. when the initialization tag IT is *needIn* and the operand stack entry for the object has the static type $init(CNAM')$. In this case $init(CNAM')$ must be introduced by rule (T-13). As mentioned in the discussion for that rule, the enclosing method must be in the class $CNAM'$. The constraint

$$(CNAM' = CNAM) \vee diSubcls(CNAM', CNAM)$$

means that the invoked `<init>` method is either in the same class as the enclosing method or in the immediate superclass of it. Analogous to the first case above, the instruction changes all occurrences of $init(CNAM')$ into $CNAM$, indicating that after the instruction (but still inside the enclosing `<init>` method) the object being initialized is regarded as having been initialized. In addition, the constraint $IT' = needNoIn$ in the rule expresses the change of the initialization tag into *needNoIn*. Rule (T-20) for `return` will use the tag to determine whether an `<init>` method really invokes another `<init>` method or not.

The constraint $NOTNULL\_VOID = void$ assures that the `<init>` method has no return type.

Note that the rule changes the component M into Mod_2 in the second case. The definition of Mod_2 will be given in Section 10.7.

The third case concerns the invocation of a usual instance method (e.g. via `super`). In this case, the constraint $widRefConvert(REFS\_RAW, CNAM)$ assures that the class $CNAM$ is a superclass of all possible classes of the object, on which the method is invoked. In addition, the constraint implicitly implies that $REFS\_RAW = REFS$ holds. Now the method may have a return type or not. the operand stack type ST' after the instruction is either $ST + NOTNULL$ or ST.

Rules (T-15), (T-16) and (T-17) are for `invokevirtual`, `invokeinterface` and `invokestatic`. They are very similar to the third case of rule (T-14). One difference is that they use the resolution functions $mResolV$, $mResolI$ and $mResolSt$, respectively, instead of $mResolSp$. In addition, rule (T-16) needs to deal with the number $BYT1$ and $BYT2$ explicitly occurring in the `invokeinterface` instruction. The `invokestatic` does not need an object, on which the method is invoked,

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{invokevirtual}\ IND1\ IND2} \\ (MNAM, CNAM, (\overline{TY_n})NOTNULL\_VOID, \_, \_) = mResolV(IND1, IND2) \\ \Phi(P) = \Pi[ST + REFS \overline{+ TYS_n}] \\ invoConvert(TYS_i, TY_i) \quad (i = 1, \ldots, n) \\ widRefConvert(REFS, CNAM) \\ MNAM \neq < \texttt{init} > \\ NOTNULL\_VOID = void \Rightarrow ST' = ST \\ NOTNULL\_VOID = NOTNULL \Rightarrow ST' = ST + NOTNULL\end{array}}{\Phi(P+3) \sqsupseteq \Pi[ST']} \quad \text{(T-15)}$$

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{invokeinterface}\ IND1\ IND2\ BYT1\ BYT2} \\ BYT1 > 0 \\ BYT2 = 0 \\ (MNAM, INAM, (\overline{TY_{BYT1-1}})NOTNULL\_VOID) = mResolI(IND1, IND2) \\ \Phi(P) = \Pi[ST + REFS \overline{+ TYS_{BYT1-1}}] \\ invoConvert(TYS_i, TY_i) \quad (i = 1, \ldots, BYT1 - 1) \\ widRefConvert(REFS, INAM) \\ MNAM \neq < \texttt{init} > \\ NOTNULL\_VOID = void \Rightarrow ST' = ST \\ NOTNULL\_VOID = NOTNULL \Rightarrow ST' = ST + NOTNULL\end{array}}{\Phi(P+5) \sqsupseteq \Pi[ST']} \quad \text{(T-16)}$$

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{invokestatic}\ IND1\ IND2} \\ (MNAM, \_, (\overline{TY_n})NOTNULL\_VOID, \_, \_) = mResolSt(IND1, IND2) \\ \Phi(P) = \Pi[ST \overline{+ TYS_n}] \\ invoConvert(TYS_i, TY_i) \quad (i = 1, \ldots, n) \\ MNAM \neq < \texttt{init} > \\ NOTNULL\_VOID = void \Rightarrow ST' = ST \\ NOTNULL\_VOID = NOTNULL \Rightarrow ST' = ST + NOTNULL\end{array}}{\Phi(P+3) \sqsupseteq \Pi[ST']} \quad \text{(T-17)}$$

Figure 15: The typing rules for other method invocation instructions

$$\frac{Prg(P) = \texttt{areturn}\ \text{or}\ \texttt{ireturn}}{\Xi[P, J + [(L', S', P')], L, S + WRD] \implies \Xi[P', J, L', S' + WRD]} \quad \text{(S-17),(S-18)}$$

$$\frac{Prg(P) = \texttt{return}}{\Xi[P, J + [(L', S', P')], L, S] \implies \Xi[P', J, L', S']} \quad \text{(S-19)}$$

Figure 16: The state transitions for return instructions

The state transition rules for return instructions are given in Figure 16. The state transition uses the return address P' stored in the current Java stack.

$$\frac{\boxed{Mth(P) = \texttt{areturn}} \quad \Phi(P) = \Pi[ST + REFS] \quad (\_, \_, (\overline{TY_n})REF, \_) = mInfo(P) \quad widRefConvert(REFS, REF)}{} \tag{T-18}$$

$$\frac{\boxed{Mth(P) = \texttt{ireturn}} \quad \Phi(P) = \Pi[ST + int] \quad (\_, \_, (\overline{TY_n})int, \_) = mInfo(P)}{} \tag{T-19}$$

$$\frac{\boxed{Mth(P) = \texttt{return}} \quad (\_, MNAM, (\overline{TY_n})void, \_) = mInfo(P) \quad MNAM = < \texttt{init} > \Rightarrow \Phi(P) = \Pi[needNoIn]}{} \tag{T-20}$$

Figure 17: The typing rules for return instructions

The typing rules for return instructions are given in Figure 17. The rules need no additional explanations. The only thing that is worth mentioning is that a `return` instruction may be used to terminate an `<init>` method. In this case, the rule checks whether the initialization tag is *needNoIn* to assure that the `<init>` method has indeed invoked another `<init>` method. Note that if the `<init>` method is in *Object*, then the tag has been set into *needNoIn* at the beginning of the method.

Note that in general, there may exist some uninitialized objects in the operand stack or local variables when a method terminates. However, there is no possibility to pass an uninitialized object to the invoking method (see Theorem 3).

10.6 Implementing `finally`-clauses

According to the OJVMS, `jsr` and `ret` instructions are control transfer instructions typically used to implement `finally` clauses in `Java`. Following the OJVMS, we call the program point, to which a `jsr` instruction jumps, a `jsr` *target*, and the code starting from a `jsr` target a *subroutine*. If no ambiguity is possible, we also call a `jsr` target a *subroutine*. Roughly speaking, a `jsr` instruction calls a subroutine and a `ret` instruction returns from a subroutine. But, formally a subroutine need not have a `ret` instruction. We use

sb to range over all `jsr` targets (i.e. subroutines) and write SB as a variable for them.

$$\frac{\begin{array}{l} \mathit{Prg}(P) = \texttt{jsr}\ BYT1\ BYT2 \\ P = (\_, \mathit{OFF}) \end{array}}{\Xi[P, S] \implies \Xi[P + \mathit{offset}(BYT1, BYT2), S + \mathit{OFF}]} \quad \text{(S-20)}$$

$$\frac{\begin{array}{l} \mathit{Prg}(P) = \texttt{ret}\ \mathit{IND} \\ P = (\mathit{COD}, \_) \end{array}}{\Xi[P, L] \implies \Xi[(\mathit{COD}, L(\mathit{IND}) + 3), L]} \quad \text{(S-21)}$$

Figure 18: The state transitions for `jsr` and `ret`

The state transitions for `jsr` and `ret` are given in Figure 18. Rule (S-20) says that a `jsr` instruction pushes the byte offset OFF of the current program point onto the operand stack and transfers control to the `jsr` target $P + \mathit{offset}(BYT1, BYT2)$.

Rule (S-21) is for `ret`. It uses a byte offset in a local variable to compute the program point following the `jsr` as the returning program point.

Typing `jsr` and `ret` is complex, since the OJVMS requires the following features:

- Not every path in a subroutine needs to reach a `ret` instruction. A subroutine implicitly terminates whenever the current method terminates.
- Subroutines may be nested: a subroutine can call another subroutine. (This feature is useful in implementing nested `finally` clauses.)
- In nested subroutines, an inner subroutine may contain a `ret` instruction that directly returns to an arbitrary outer subroutine.
- During the execution, a returning program point can never be used more than once by a `ret` instruction. Furthermore, at the outer program point, to which a `ret` instruction in an inner subroutine directly returns, no returning program point for a subroutine between the inner and the outer subroutine should still be able to be used as a returning program point.

Technically, the mechanism should be more complex, since the OJVMS still takes three additional situations into account. First, the implementation of a `finally` clause often needs to be reachable from different execution paths. Second, different execution paths often have to use a common local variable to hold their own contents that are incompatible to each other. Third, the content stored in a local variable in an execution path before the execution of a `finally` clause may need to be used after the execution of the `finally` clause. As an example for all these three situations, we can consider the implementation of a `try-catch-finally` clause. More concretely, the `finally` clause needs to be reachable from the end of the `try` clause and from the beginning of the `catch` clause, the `try` clause needs to store a return integer value in a local variable for use after the execution of the `finally` clause, but the `catch` clause stores an exception in the same local variable for use after the execution of the `finally` clause as well.

The problem is that since a common local variable may hold incompatible contents, as described in the second situation above, the usual typing rules in our formal specification would force the local variable in and, in particular, after the `finally` clause to have the type *unusable*. Therefore a use of the local variable in an individual execution path after the `finally` clause, as described in the third situation, would be impossible.

To solve the problem, the OJVMS suggests to change the usual typing process such that in an execution path, if a local variable is not modified or accessed in a `finally` clause, then its type after the execution of the `finally` clause should be the same as before the execution of the `finally` clause. Thus we need a mechanism to record the local variables that are modified or accessed within a `finally` clause. The component *mod* in a program point type has been reserved for this purpose. Now we formally define what a component *mod* is:

- First, we build a set *grf* of pairs of `jsr` targets, representing a directed acyclic graph.
- Then we build a set *csb* of `jsr` targets.
- Finally, a component *mod* is a mapping such that $\mathcal{D}om(mod) = grf \cup csb$, $mod(sb, sb')$ for $(sb, sb') \in grf$ and $mod(sb)$ for $sb \in csb$ are sets of indices of local variables.

Intuitively, a pair (sb, sb') in a *grf* should denote a call of the subroutine sb' inside the subroutine sb, and *grf* should contain nested non-recursive subroutine calls that may reach the current program point. A set *grf* need not be a tree, since more than one subroutine may contain a call of the same subroutine and one subroutine may contain calls of more than one subroutine. A set *csb* should contain current subroutines, i.e. those subroutines that contain the current program point. The set $mod(sb, sb')$ for $(sb, sb') \in grf$ should contain the indices of all local variables that may be modified or accessed in an execution path from sb to sb', and $mod(sb)$ for $sb \in csb$ those from sb to the current address.

We define the following notations:

$$\begin{aligned}
nod(mod) &:= \{sb \mid (sb, _) \text{ or } (_, sb) \text{ or } sb \in \mathcal{D}om(mod)\} \\
grf(mod) &:= \{(sb, sb') \mid (sb, sb') \in \mathcal{D}om(mod)\} \\
csb(mod) &:= \{sb \mid sb \in \mathcal{D}om(mod)\}
\end{aligned}$$

We define that $mod \sqsupseteq mod'$ holds if and only if $grf(mod) \supseteq grf(mod')$ and $csb(mod) \supseteq csb(mod')$ hold, $mod(sb, sb') \supseteq mod'(sb, sb')$ holds for each $(sb, sb') \in grf(mod')$ and $mod(sb) \supseteq mod'(sb)$ holds for each $sb \in csb(mod')$.

The typing rules for `jsr` and `ret` are given in Figure 19. We first consider rule (T-21). The rule defines only one constraint at the program point P

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{jsr}\ BYT1\ BYT2} \\ \Phi(P) = \Pi[ST, M] \\ SB = P + \mathit{offset}(BYT1, BYT2) \\ SB \notin nod(M)\end{array}}{\begin{array}{l}\Phi(SB) \sqsupseteq \Pi[ST + sbr(SB), \\ \qquad M_{|grf(M)} \cup \{(sb, SB) \mapsto M(sb) \mid sb \in csb(M)\} \cup \{SB \mapsto \emptyset\}]\end{array}} \quad \text{(T-21)}$$

$$\frac{\begin{array}{l}\boxed{Mth(P) = \texttt{ret}\ IND} \\ \Phi(P) = \Pi[LT] \\ LT(IND) = sbr(\_) \\ \forall P' \forall IND' \forall \Pi' \forall LT'.(\ (Mth(P') = \texttt{ret}\ IND' \ \wedge\ P' \neq P \ \wedge\ \Phi(P') = \Pi'[LT']) \\ \qquad \Rightarrow LT(IND) \neq LT'(IND')\ )\end{array}}{} \quad \text{(T-22)}$$

$$\frac{\boxed{\begin{array}{l}Mth(P) = \texttt{ret}\ IND \\ \Phi(P) = (LT, ST, IT, M) \\ Mth(P') = \texttt{jsr}\ BYT1\ BYT2 \\ SB = P' + \mathit{offset}(BYT1, BYT2) \\ LT(IND) = sbr(SB) \\ \Phi(P') = \Pi'[LT', ST', M']\end{array}}}{\begin{array}{l}\Phi(P'+3) \sqsupseteq (LT'[j \mapsto invld(LT(j), sbrs(SB, M)) \mid\ j \in mlvs(SB, M)], \\ \qquad invld(ST, sbrs(SB, M)), IT, \\ \qquad reachMod(M, \{sb \mid (sb, SB) \in \mathcal{D}om(M)\}) \cup \\ \qquad \{sb \mapsto M'(sb, SB) \cup mlvs(SB, M) \mid (sb, SB) \in \mathcal{D}om(M)\})\end{array}} \quad \text{(T-23)}$$

Figure 19: The typing rules for `ret` and `jsr`

of a `jsr` instruction, namely $SB \notin nod(M)$, which assures that the called subroutine SB is not called recursively. At the beginning of the subroutine SB, the new M records the addition of the edges (sb, SB) representing the calls of SB inside all old current subroutines in $csb(M)$, and elimination of the old current subroutines in $csb(M)$ and addition of the new current subroutine SB, where $SB \mapsto \emptyset$ denotes that no local variables have been modified or accessed since the beginning of the new current subroutine SB.

Rule (T-22) is for `ret`. The constraint $LT(IND) = sbr(\_)$ assures that the local variable IND holds a byte offset. The constraint $\forall P' \forall IND' \forall \Pi' \forall LT'. \cdots$ assures that the method Mth has at most one `ret` instruction for the same subroutine. This is not a serious restriction, since whenever two `ret` instructions are needed, one can always write the first `ret` and at the place of the second `ret` a `goto` instruction to the first `ret`.

Rule (T-23) introduces constraints for the program type at the returning program point $P' + 3$, to which a `ret` at P returns, where the calling `jsr` of the subroutine is at P'.

The formulation of rule (T-23) uses several new auxiliary functions.

The first auxiliary function computes the set of the indices of all local variables that may be modified or accessed in an execution path from a given program point to the current program point. For the component mod in a program point type and a subroutine $sb \in nod(mod)$, we define

$$mlvs(sb, mod) := \begin{cases} \bigcup_{(sb,sb') \in grf(mod)} (mod(sb, sb') \cup mlvs(sb', mod)) \\ \qquad\qquad\qquad\qquad\qquad \text{if } sb \notin csb(mod) \\ mod(sb) \qquad\qquad\qquad\quad \text{if } sb \in csb(mod) \end{cases}$$

The term $mlvs(SB, M)$ in rule (T-23) is a set containing the indices of all local variable that may be modified or accessed from SB to P.

The second auxiliary function computes all subroutines called from the call of a given outer subroutine to the current subroutine. For the component mod in a program point type and a subroutine $sb \in nod(mod)$, we define

$$sbrs(sb, mod) := \begin{cases} \bigcup_{(sb,sb') \in grf(mod)} \{sb\} \cup sbrs(sb', mod) & \text{if } sb \notin csb(mod) \\ \{sb\} & \text{if } sb \in csb(mod) \end{cases}$$

In order to change all subroutine types of those subroutines in a given set of subroutines E into invalid subroutine types, we define the following function $invld(any, E)$:

$$invld(any, E) := \begin{cases} invldsbr & \text{if } any = sbr(sb) \text{ and } sb \in E \\ any & \text{otherwise} \end{cases}$$

Note that any in the second line can be an arbitrary static type.

For convenience, we lift the function $invld$ to operand stack types:

$$invld([any_0, \cdots, any_m], E) := [invld(any_0, E), \cdots, invld(any_m, E)]$$

In order to compute the part of a mod, which is reachable to a subroutine in a given set of subroutines E, we define the following function $reachMod(mod, E)$:

$$reachMod(mod, E) := \begin{cases} \{(sb, sb') \mapsto mod(sb, sb') \mid (sb, sb') \in \mathcal{D}om(mod), sb' \in E\} & \\ \quad \cup\, reachMod(mod, \{sb \mid (sb, sb') \in \mathcal{D}om(mod), sb' \in E\}) & \text{if } E \neq \emptyset \\ \emptyset & \text{if } E = \emptyset \end{cases}$$

In rule (T-23), the applicability conditions $\Phi(P) = (LT, \cdots)$, $Mth(P') =$ `jsr` $BYT1\ BYT2$, $SB = P' + \mathit{offset}(BYT1, BYT2)$ and $LT(IND) = sbr(SB)$ assure that the `ret` at P causes the subroutine SB to return to the next program point $P' + 3$ of the calling `jsr` at P'. Note that the constraint $\forall P' \forall IND' \forall \Pi' \forall LT'. \cdots$ in rule (T-22) enforces that there exists at most one P for a `jsr` at P' in rule (T-23).

Rule (T-23) expresses the following relationship between the program types at P, P' and $P' + 3$, where the `jsr` at P' calls a subroutine SB, the `ret` at P returns from the subroutine SB to $P' + 3$:

- If a local variable is definitely not modified or accessed from SB to P, then its static type at $P' + 3$ covers that at P'; otherwise, i.e. if the local variable may be modified or accessed from SB to P, then its static type at $P' + 3$ covers that at P, except that if its static type at P is a subroutine type for a subroutine possibly called from SB to P, then its static type at $P' + 3$ covers $invldsbr$.
- The operand stack type at $P' + 3$ covers that at P, except that if an operand stack entry at P has a subroutine type for a subroutine called from SB to P, then the static type of the entry at $P' + 3$ covers $invldsbr$.
- The initialization tag at $P' + 3$ covers that at P.
- The subroutines called until $P' + 3$ include all those called until P but not from SB to P. The local variables possibly modified or accessed from the call of a current subroutine to $P' + 3$ include those from the call of the same current subroutine to SB plus all those possibly modified or accessed from SB to P.

A final tricky point is that although the `ret` instruction in rule (T-23) accesses all those local variables that have a subroutine type for a subroutine called from P' to P, the typing rule need not treat this explicitly. The reason is that the indices of these variables are all contained in the set $mlvs(SB, M)$ in rule (T-23). In fact, if a local variable holds a program point of a `jsr` instruction between SB and P, then the program point must be stored in the local variable by an `astore` instruction between SB and P. By the typing rule for `astore` (see the discussion below) and the definition of $mlvs$, the index of the local variable must be included in the set $mlvs(SB, M)$.

10.7 On the instructions that modify or access local variables

Now it is time to give the precise definitions of the term Mod' in Figure 4, the term mod_0 in Figure 13 and the terms Mod_1 and Mod_2 in 14.

We first consider the typing rules in Figure 4. Given the notations in Figure 4, we formally define

$$Mod' := M_{|grf(M)} \cup \{sb \mapsto M(sb) \cup \{IND\} \mid sb \in csb(M)\}$$

The typing rule in Figure 13 introduces the program point type for the starting program point of a method. We define $mod_0 := \emptyset$.

Now we consider rule (T-14) for `invokespecial` in Figure 14. Since the first two cases in rule (T-14) consider the initialization of a raw object, we regard all those local variables whose contents reference the raw object as being modified. Given the notations in Figure 14, we formally define

$$\begin{aligned} Mod_1 := {} & M_{|grf(M)} \cup \\ & \{sb \mapsto M(sb) \cup \{i \mid unin(P', CNAM) = LT(i)\} \mid sb \in csb(M)\} \\ Mod_2 := {} & M_{|grf(M)} \cup \{sb \mapsto M(sb) \cup \{i \mid init(CNAM) = LT(i)\} \mid sb \in csb(M)\} \end{aligned}$$

Note that the third case in rule (T-14) does not deal with initialization of a raw object, thus does not cause the extension of the M.

11 Examples

In this section we use real methods to illustrate how to check whether a given method has a given program type.

For notational simplicity, some instructions are abbreviated as follows:

- Each instruction

  $$opcode\ byt1\ byt2$$

  at the program point pp with $opcode \in \{\texttt{if\_acmeq}, \texttt{if\_icmeq}, \texttt{goto}, \texttt{jsr}\}$ and $pp = (\_, off)$ is abbreviated as

  $$opcode\ n$$

  with $n = off + (byt1 * (2^8)) + byt2$.
- Each instruction

  $$opcode\ ind1\ ind2$$

  with $opcode \in \{$`getfield`,`putfield`,`new`,`invokespecial`,`invokevirtual`, `invokeinterface`, `invokestatic`$\}$ is abbreviated as

  $$opcode\ \#n$$

  with $n = (ind1 * (2^8)) + ind2$. The instruction

  $$\texttt{invokeinterface}ind1\ ind2\ byt\ 0$$

  is abbreviated as `invokeinterface` $\#n$ with $n = (ind1 * (2^8)) + ind2$.

Figure 20 gives the type checking for the first method. A row in Figure 20 contains a program point, i.e. an instruction, in the given method, the program point type in the given program type at the program point, the typing rule applied at the program point and all possible successor program points with respect to the rule. Since the method does not deal with any subroutines or instance initializations, we consider only the local variable table type and the operand stack type in a program point type.

We assume that the declaration of the method `void m(J1,J2)` in Figure 20 is contained in a class C. Furthermore, we assume that `J1` and `J2` are two interfaces, and the entry at the index `#17` in the constant pool references a method in a superinterface of `J1` and `J2`, which takes no parameters and yields no result. At the program point `13`, the static type of the top entry in the operand stack needs to be represented as a set, since the top entry may be the first or second actual parameter and the interfaces `J1` and `J2` need not have one smallest common superinterface. Rule (T-16) is applied at the program point `13`, where the constraint $widRefConvert(REFS, INAM)$ in the rule assures that the invoked method must exist in a superinterface of `J1` and `J2`.

The method	LT	ST	Rule	Successors
`Method void m(J1,J2)`			(T-13)	`0`
`0 aconst_null`	$[C, J1, J2]$	$[]$	(T-5)	`1`
`1 aload 1`	$[C, J1, J2]$	$[null]$	(T-1)	`3`
`3 if_acmpeq 11`	$[C, J1, J2]$	$[null, J1]$	(T-10)	`6, 11`
`6 aload 1`	$[C, J1, J2]$	$[]$	(T-1)	`8`
`8 goto 13`	$[C, J1, J2]$	$[J1]$	(T-12)	`13`
`11 aload 2`	$[C, J1, J2]$	$[]$	(T-1)	`13`
`13 invokeinterface #17`	$[C, J1, J2]$	$[\{J1, J2\}]$	(T-16)	`18`
`18 return`	$[C, J1, J2]$	$[]$	(T-20)	

Figure 20: A method containing an interface method invocation

The second example in Figure 21 shows the use of subroutine types. The method contains two `jsr` instructions at `0` and `9` calling the subroutine `13`. The subroutine `13` contains a `jsr` at `15` calling the (inner) subroutine `18`, and the subroutine `18` directly returns to the corresponding calling `jsr` of the (outer) subroutine `13`, i.e. to `3` or `12`. After the return, i.e. at `3` and `12`, the subroutine types $sbr(13)$ and $sbr(18)$ are changed into $invldsbr$. The local variable 1 has different static types at the two calling `jsr`, i.e. at `0` and `9`. Since the local variable 1 is not modified or accessed in the subroutine `13`, after the return of the subroutine, i.e. at `3` and `12`, the static type of the local variable 1 is the same as that at the calling `jsr`, i.e. at `0` and `9`.

The method	LT	ST	Rule	Successors
`Method void m()`			(T-13)	0
`0 jsr 13`	$[C, unusable, unusable]$	$[]$	(T-21)	13
`3 astore 2`	$[C, unusable, invldsbr]$	$[invldsbr]$	(T-3)	5
`5 aload 0`	$[C, unusable, invldsbr]$	$[]$	(T-1)	7
`7 astore 1`	$[C, unusable, invldsbr]$	$[C]$	(T-3)	9
`9 jsr 13`	$[C, C, invldsbr]$	$[]$	(T-21)	13
`12 return`	$[C, C, invldsbr]$	$[invldsbr]$	(T-20)	
`13 astore 2`	$[C, unusable, unusable]$	$[sbr(13)]$	(T-3)	15
`15 jsr 18`	$[C, unusable, sbr(13)]$	$[]$	(T-21)	18
`18 ret 2`	$[C, unusable, sbr(13)]$	$[sbr(18)]$	(T-22)	
			(T-23)	3
			(T-23)	6

Figure 21: A method containing subroutines

12 Static well-typedness vs. runtime properties

The OJVMS requires that the type-correctness of nearly all runtime uses of data is checked statically. In our formal specification, which considers a subset of the JVM, we can formally prove that if a program is statically well-typed, then all runtime data to be used will definitely have correct types. For doing this, we first need to define precisely what are the types of runtime data.

12.1 Tags of runtime data

In the previous sections we have often informally mentioned the types of runtime data. Two examples are as follows:

– In Section 10.2 we mentioned that a `new` instruction creates an object. This informally implies that the created datum is a reference of an object.
– In Section 10.6 we mentioned that a `jsr` instruction pushes a byte offset onto the operand stack.

However, the problem is that both an object reference and a byte offset are one-word wide data in our constraint domain. In other words, the type of a datum cannot be determined by the datum itself. Thus we need an additional mechanism to explicitly determine the type of a datum.

The mechanism can be built in two steps: first, we define the possible types of runtime data; second, we extend the state transition relation to define the types of the contents in the local variables and the operand stack.

A relatively simple set of possible types of runtime data, called *tags*, is defined as follows:

$$tag ::= cnam \mid null \mid int \mid addr \mid undefined$$

Intuitively, the tag *cnam* should be the tag of the reference of each object of the class *cnam*, *null* that of the special reference *null*, and *int* that of each element of the primitive type *int*. As mentioned before, we need to deal with the byte offset of a `jsr`. Thus we introduce the tag *addr* for all byte offsets. The tag *undefined* indicates that the content of a local variable or an operand stack entry has not been explicitly defined by an instruction in the execution so far. We use *tag* to range over all tags.

Note that the above set of tags is a relatively simple one, since they do not contain anything to express that an object is a raw object or an offset is of a special subroutine type. In fact, there are no problems to do that, except that the definition of the types of the contents in the local variables and the operand stack would become more complicated. We consider the above simple set due to space limits in this chapter.

To record the type of the content of each local variable and each operand entry, lists of tags of the form $[tag_0, \cdots, tag_n]$ with $n \geq -1$ are introduced. We define $[tag_0, \cdots, tag_n](k) := tag_k$ if $0 \leq k \leq n$, $[tag_0, \cdots, tag_n](k) := failure$ otherwise. A list of the above form is called a *local variable state tag* if it consists of the types of the contents in all local variables; it is called an *operand stack state tag* if it consists of the types of the contents of an operand stack.

For readability, we use *lvstag* to range over all local variable state tags, and *stktag* over all operand stack state tags. For notational simplicity, we write LG as a variable for the sort *lvstag*, and SG for the sort *stktag*.

A local variable state tag and an operand stack state tag do not record the type of an object that is held by a field of another object but not directly by a local variable or an operand stack entry. Thus we still need to introduce a *class record* as a mapping $\{\overline{obj_n \mapsto cnam_n}\}$. A class record as above maps all elements other than $\overline{obj}_n$ to a special value *failure*. We use *classof* to range over all concrete class records and C as a variable for the sort *classof*.

In order to record the local variable state tags and the operand stack state tags for the methods stored in a Java stack, we define a *Java stack tag* as a list consisting of entries of the form $(lvstag, stktag)$. We use *jstktag* to range over all Java stack tags and use JG as a variable of the sort *jstktag*.

We define a *program state tag* as a tuple $(jstktag, classof, lvstag, stktag)$ and in the rest of the chapter still use *statag* to range over all program state tags.

Finally, we define that an *extended program state* is a pair $(stat, statag)$, where $stat = (pp, jstk, lvs, stk, hp)$, $statag = (jstktag, classof, lvstag, stktag)$, $size(jstk) = size(jstktag)$, $size(lvs) = size(lvstag)$ and $size(stk) = size(stktag)$ hold.

Now we extend the state transition rules in Section 10. Let us call an extended state transition rule an *extended rule* and an original state transition rule an *extended rule* in this section.

In order to ensure that the extended rule relation does not affect the original state transition relation, we require that if an original rule in Section 10 is of the form

$$\frac{\mathit{Premises}}{\mathit{Stat} \implies \mathit{Stat}'}$$

then the extended rule obtained from it is always of the form

$$\frac{\mathit{Premises}}{\mathit{Stat}, \mathit{Statag} \implies \mathit{Stat}', \mathit{Statag}'}$$

satisfying that

- $\mathcal{FV}(\mathit{Statag}') \subseteq \mathcal{FV}(\mathit{Statag})$
- for every two program states stat and stat' and every extended program state $(\mathit{stat}, \mathit{statag})$, if there is a substitution σ such that $\mathcal{D}om(\sigma) = \mathcal{FV}(\mathit{Premises}) \cup \mathcal{FV}(\mathit{Stat} \implies \mathit{Stat}')$, $\mathit{stat} = \sigma(\mathit{Stat})$, $\mathit{stat}' = \sigma(\mathit{Stat}')$ and $\sigma(\mathit{Premises})$ hold, then there is a substitution σ' such that $\mathcal{D}om(\sigma') = \mathcal{D}om(\sigma) \cup \mathcal{FV}(\mathit{Statag})$ and $\sigma'(\mathit{Statag}) = \mathit{statag}$ hold, and $(\mathit{stat}', \sigma'(\mathit{Statag}'))$ is an extended program state.

For notational simplicity, we always omit the *Premises*-, *Stat*- and Stat'-parts in the definition of an extended rule in this section. Note that the *Statag*- and Statag'-parts may contain variables occurring in *Stat*- and Stat'-parts.

In many extended rules, the Java stack tags are not changed and the local variable state tags (or the operand stack state tags) are changed in a completely analogous way as the local variable states (or the operand stack states, respectively). The extended rules for `aload` and `new` are two such extended rules. We give their definitions in Figure 22 and omit the explicit presentation of other such extended rules due to space constraints.

$$\frac{}{(JG, C, LG, SG) \implies (JG, C, LG, SG + LG(\mathit{IND}))} \tag{S'-1}$$

$$\frac{}{(JG, C, LG, SG) \implies (JG, C[\mathit{OBJ} \mapsto \mathit{CNAM}], LG, SG + \mathit{CNAM}]} \tag{S'-8}$$

Figure 22: The extended rules for `aload` and `new`

Figure 23 contains the extended rule for `getfield`. The rule is slightly tricky, since the way to get the tag of the loaded content depends on whether the loaded content is an object or not. If it is an object, then the tag should be obtained from the class record *classof* in the program state. If it is a value of a primitive type, then the tag should the primitive type. (In this chapter the only primitive type is the type *int*.) To model this, we define the following auxiliary function, which yields the tag of the content held by the field *fnam* of the type *notnull* in the object *obj*.

$$seltag(fnam, notnull, obj, hp, classof) := \begin{cases} classof(hp(obj)(fnam)) & \text{if } notnull \text{ is a } cnam \\ int & \text{if } notnull \text{ is } int \\ failure & \text{otherwise} \end{cases}$$

$$\frac{}{\begin{array}{l}(JG, C, LG, SG) \\ \Longrightarrow\ (JG, C, LG, SG + seltag(FNAM, NOTNULL, OBJ, H, C))\end{array}} \qquad \text{(S'-6)}$$

Figure 23: The extended rule for `getfield`

Rules (S-13), (S-14) and (S-15) for method invocations change the Java stack states. Thus their extended rules change the Java stack tags. Since these extensions are very similar, we present only one of them. The situation is similar for rules (S-17) and (S-19). Thus we present only one of the two extended rules. Figure 24 these two extended rules, where UD^k stands for a list $[undefined, \cdots, undefined]$ consisting of k times *undefined*.

$$\frac{}{\begin{array}{l}(JG, C, LG, SG + TAG_0 \overline{+ TAG_n}) \\ \Longrightarrow\ (JG + (LG, SG), C, UD^{MXL}[i \mapsto TAG_i \mid 0 \leq i \leq n], [])\end{array}} \qquad \text{(S'-13)}$$

$$\frac{}{\begin{array}{l}(JG + (LG', SG'), C, LG, SG + TAG \\ \Longrightarrow\ (JG + (LG', SG'), C, LG', SG' + TAG]\end{array}} \qquad \text{(S'-17)}$$

Figure 24: The extended rules for `invokespecial` and `areturn`

$$\frac{}{(JG, C, LG, SG) \Longrightarrow (JG, C, LG, SG + addr)} \qquad \text{(S'-20)}$$

$$\frac{}{(JG, C, LG, SG) \Longrightarrow (JG, C, LG, SG)} \qquad \text{(S'-21)}$$

Figure 25: The extended rules for `jsr` and `ret`

Figure 25 contains the extended rules for `jsr` and `ret`. Note that in rule (S'-21), the program state tag does not change at all. The intuition is that a `ret` may change some the validity of some byte offsets. However, since we consider only a simple tag *addr* for byte offsets, this intuition cannot be reflected. (As mentioned, the simple tag *addr* could be replaced by a family of tags indexed by all subroutines. But we do not consider them here.)

12.2 The concepts for runtime type safety

To model the correctness of a tag *tag* with respect to a static type *any*, we formally define a relation *correct* by:

$$\begin{array}{ll}
correct(null, refs) & \\
correct(cnam, refs) & \text{if } widRefConvert(cnam, refs) \\
correct(cnam, unin(_, cnam)) & \\
correct(cnam, init(cnam')) & \text{if } widRefConvert(cnam, cnam') \\
correct(int, int) & \\
correct(addr, sbr(_)) & \\
correct(undefined, unusable) & \\
correct(tag, any) & \text{if } correct(tag, any') \text{ and } any \sqsupseteq any'
\end{array}$$

We also define that $correct(lvstag, lvsty)$ holds if and only if $size(lvstag) = size(lvsty)$ and $correct(lvstag(i), lvsty(i))$ for all $i = 0, \ldots, size(lvstag)$, and that $correct(stktag, stkty)$ holds if and only if $size(stktag) = size(stkty)$ and $correct(stktag(i), stkty(i))$ for all $i = 0, \ldots, size(stktag)$.

For a heap *hp* and a class record *classof*, we define that $correct(hp, classof)$ holds if and only if the following conditions are true:

1. $\mathcal{D}om(hp) \subseteq \mathcal{D}om(classof)$.
2. For each $obj \mapsto rec \in hp$, if $(fnam, notnull) \in allFields(classof(obj))$, then $fnam \in \mathcal{D}om(rec)$.
3. For each $obj \mapsto rec \in hp$ and $(fnam, notnull) \in allFields(classof(obj))$, if $notnull = ref$, then $rec(fnam) \in \mathcal{D}om(classof)$.
4. For each $obj \mapsto rec \in hp$ and $(fnam, notnull) \in allFields(classof(obj))$, if $notnull = ref$, then $widRefConvert(classof(rec(fnam)), notnull)$.

Intuitively, condition 1 says that *classof* can determine the class of each object in *hp*. Condition 2 assures that an object in *hp* always contains all fields required by its class. Condition 3 assures that if an object in *hp* contains a field whose type is a class or an interface, then the field holds an object, whose class can be determined by *classof*. Condition 4 says that the class of the object held by the field in condition 3 is a subtype of the class or interface of the field.

Note that if $notnull \neq ref$, i.e. if $notnull = int$, then conditions 3 and 4 have no effects. Thus one might wonder why we do not define a condition constraining $rec(fnam)$. The intuition is that if the runtime type of a datum is a primitive type, then the runtime type is always the static type. Thus for $(fnam, int) \in allFields(classof(obj))$ and $obj \mapsto rec \in hp$, the content $rec(fnam))$ is always an integer of the type *int*. Hence such a condition is useless.

12.3 Runtime properties

From now on, we assume that the program *Prg* has a program type *prgty*.

Formally we define an arbitrary *execution* of *Prg* as

$$(stat_1, statag_1) \Longrightarrow (stat_2, statag_2) \Longrightarrow \cdots$$

where each $(stat_i, statag_i)$ for $i = 1, 2, 3, \cdots$, are extended program states, $stat_1$ is of the form $(pp_1, [], \cdots)$ and $Prg(pp_1)$ is of the form `invokestatic` $\cdots$.

We use $(stat_1, statag_1) \Longrightarrow^* (stat_h, statag_h)$ with $h \geq 1$ to denote a zero or more step execution

$$(stat_1, statag_1) \Longrightarrow \cdots \Longrightarrow (stat_h, statag_h)$$

For the rest of the chapter, we assume that

- $stat_i = (pp_i, jstk_i, lvs_i, stk_i, hp_i)$ for all $i = 1, 2, 3, \ldots$,
- $statag_i = (jstktag_i, classof_i, lvstag_i, stktag_i)$ for all $i = 1, 2, 3, \ldots$, and
- $prgty(pp_i) = ptty_{pp_i} = (lvsty_{pp_i}, stkty_{pp_i}, intag_{pp_i}, mod_{pp_i})$ for all $i = 1, 2, 3, \ldots$. Note that $i \neq j$ does not imply that $pp_i \neq pp_j$.

Now we give some lemmas and theorems. Proofs are omitted due to space limits.

The first theorem states the runtime type safety.

Theorem 1. *In the execution* $(stat_1, statag_1) \Longrightarrow (stat_2, statag_2) \Longrightarrow \cdots$, *if* $correct(lvstag_1, lvsty_{pp_1})$, $correct(stktag_1, stkty_{pp_1})$ *and* $correct(hp_1)$ *hold, then* $correct(lvstag_i, lvsty_{pp_i})$, $correct(stktag_i, stkty_{pp_i})$ *and* $correct(hp_i)$ *hold for all* $i = 1, 2, \cdots$.

The proof follows from an induction on the length of the execution using the extended rules and typing rules.

A practical consequence of Theorem 1 is as follows:

Corollary 2. *An offset cannot be manipulated by an instruction described in our formal specification except:*

1. *It can be created and stored onto the operand stack by a* `jsr`.
2. *It can be manipulated in the operand stack by the stack manipulation instruction* `dup`.
3. *It can be stored from the operand stack in a local variable by an* `astore`.
4. *In a local variable, it can be used to compute the return address by a* `ret`.

Now let us consider raw objects and instance initialization methods. The following theorems can either be proved using a set of tags for runtime data that is more refined then the current one, or by a careful analysis of all possible executions. Note that Theorems 3 and 4 are not completely trivial, since a method may pass values via the heap.

Theorem 3. *Assume that a method invokes another method. Then the invoked method can never pass a raw object back to the invoking method.*

Theorem 4. *Assume that a method invokes another method that is not an* `<init>`*. Then the invoking method can never pass a raw object to the invoked method.*

It is very easy to show how an instance initialization method invokes another instance initialization method.

Theorem 5. *If an instance initialization method is not in class Object, then a fragment of an execution path from the starting address to a* `return` *instruction of the method always includes exactly one invocation of an instance initialization method of the same class or the immediate superclass on the object being initialized. If the instance initialization method is in class Object, then the fragment includes no invocations of an instance initialization method on the object being initialized.*

Now we can state when the static type of a local variable or an operand stack entry ensures that it contains a raw object.

Theorem 6. *Assume that* $(stat_1, statag_1) \Longrightarrow (stat_2, statag_2) \Longrightarrow \cdots$ *is an execution and* $\mathcal{X} \in \{lvs_h, stk_h\}$ *and* $\mathcal{XT} \in \{lvsty_h, stkty_h\}$ *with* $h \geq 1$ *are such that* $\mathcal{X}$ *is* lvs_h *if and only if* $\mathcal{XT}$ *is* $lvsty_h$ *(and thus* $\mathcal{X}$ *is* stk_h *if and only if* $\mathcal{XT}$ *is* $stkty_h$*).*

- *If* $\mathcal{XT}(k) = unin(pp, cnam)$ *holds for some* k*,* pp *and* $cnam$*, then* $\mathcal{X}(k)$ *contains a reference to an uninitialized object of the class* $cnam$ *created by a* `new` *at* pp*.*
- *If* $\mathcal{XT}(k) = init(cnam)$ *holds for some* k *and* $cnam$*, then* $\mathcal{X}(k)$ *contains a reference to an object of* $cnam$ *that is being initialized inside an* `<init>` *and has not been initialized by another* `<init>`*.*

The following lemma shows that it is impossible for two different local variables/operand stack entries at a program point to have the same static type $unin(pp, cnam)$ for some pp and $cnam$ but hold references to different uninitialized objects. In fact, the lemma states the correctness of the typing rule for `invokespecial` on an instance initialization method, i.e., that if an object in a local variable/operand stack entry with the static type $unin(pp, cnam)$ is initialized, then all occurrences of $unin(pc, cnam)$ can be replaced by $cnam$.

Lemma 7. *Assume that* $(stat_1, statag_1) \Longrightarrow (stat_2, statag_2) \Longrightarrow \cdots$ *is an execution and* $\mathcal{X}, \mathcal{Y} \in \{lvs_h, stk_h\}$ *and* $\mathcal{XT}, \mathcal{YT} \in \{lvsty_h, stkty_h\}$ *with* $h \geq 1$ *such that* $\mathcal{X}$ *is* lvs_h *if and only if* $\mathcal{XT}$ *is* $lvsty_h$*, and* $\mathcal{Y}$ *is* lvs_h *if and only if* $\mathcal{YT}$ *is* $lvsty_h$*. Then the following conditions cannot hold at the same time for the indices* k *and* k'*:*

- $\mathcal{XT}(k) = \mathcal{YT}(k') = unin(pp, cnam)$ *holds for certain pp and cnam.*
- $\mathcal{X}(k)$ *and* $\mathcal{Y}(k')$ *contain references to different uninitialized objects created by the same* `new` *at pp.*

Now we know that if a memory location has a class as a static type, then it always holds initialized object or *null*.

Theorem 8. *Assume that* $(stat_1, statag_1) \implies (stat_2, statag_2) \implies \cdots$ *is an execution and* $\mathcal{X} \in \{lvs_h, stk_h\}$ *and* $\mathcal{XT} \in \{lvsty_h, stkty_h\}$ *with* $h \geq 1$ *such that* $\mathcal{X}$ *is* lvs_h *if and only if* $\mathcal{XT}$ *is* $lvsty_h$*. If* $\mathcal{XT}(k) = cnam$ *holds for some k and cnam, then* $\mathcal{X}(k)$ *contains a reference to an initialized object of cnam or null.*

The typing rules for an instruction specify precisely how the instruction behaves on an uninitialized object. The following theorem summarizes some of the results:

Theorem 9. *1. A reference to an uninitialized object cannot be used in an instruction described in our formal specification except it is* `dup aload`*,* `astore` *or* `invokespecial`*. In the case of* `invokespecial`*, the method must be* `<init>`*, the object is the one being initialized and must be of the same class as the* `<init>`*.*

2. Inside a method `<init>` *that is not declared in the class Object, there must be a call to another* `<init>` *on the object being initialized via an* `invokespecial`*, where the called* `<init>` *is declared in the same class as or in an immediate superclass of that of the calling* `<init>`*. Before this call, the object being initialized cannot be used in an instruction described in our formal specification except it is* `dup`*,* `aload` *or* `astore`*.*

13 Conclusion

We have shown a formal specification of a substantial subset of JVM instructions. The formal specification clarifies some ambiguities and incompleteness and removes some (in our view) unnecessary restrictions in the description of the official Java Virtual Machine Specification [10].

Finally, it is worth mentioning that our study on the semantics of the JVM in this chapter led to the discovery of a possibility of writing a constructor that invoked no other constructor in the JDK 1.1.4 implementation of the JVM, which is clearly an implementation bug with respect to the official Java Virtual Machine Specification (page 122).

Acknowledgement

Thanks to Gilad Bracha and David von Oheimb for clarifying comments and useful feedback, and Masami Hagiya for pointing out an error in the description.

References

1. A. Aho, R. Sethi, and J. Ullman. *Compilers — Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.
2. R. Cohen. The Defensive Java Virtual Machine specification. Technical report, Computational Logic inc., 1997.
3. D. Dean. The security of static typing with dynamic linking. In *Proc. 4th ACM Conf. on Computer and Communications Security*. ACM, 1996.
4. S. Dossopoulou and S. Eisenbach. Java is type safe — probably. In *Proc. 11th European Conf. on Object-Oriented Programming*, pages 389–418. Springer-Verlag LNCS 1241, 1997.
5. S. Freund and J. Mitchell. A type system for object initialization in the java bytecode language. Presneted at Int. Workshop on Security and Languages, Oct. 1997.
6. 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`.
7. A. Goldberg. A specification of Java loading and bytecode verification. 1997.
8. J. Gosling, B. Joy, and G. Steele. *The Java^{TM} Language Specification*. Addison-Wesley, 1996.
9. M. Hagiya. On a new method fot dataflow analysis of Java Virtual Machine subroutines. 1998.
10. T. Lindholm and F. Yellin. *The Java^{TM} Virtual Machine Specification*. Addison-Wesley, 1996.
11. T. Nipkow and D. von Oheimb. Java$_{\ell ight}$ is type-safe — definitely. In *Proc. 25st ACM Symp. Principles of Programming Languages*, 1998.
12. Z. Qian. A formal specification of Java^{tm} Virtual Machine instructions. Technical report, FB Informatik, Universität Bremen, September 1997. Revised version to appear June 1998.
13. V. Saraswat. Java is not type-safe. Technical report, AT&T Research, 1997.
14. E. Sirer, S. McDirmid, and B. Bershad. A Java system security architecture. `http://kimera.cs.washington.edu/`, 1997.
15. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *Proc. 25st ACM Symp. Principles of Programming Languages*, 1998.
16. D. Syme. Proving Java type soundness. Technical report, University of Cambridge Computer Laboratory, 1997.