A Formal Specification of Java™ Virtual Machine Instructions

Zhenyu Qian
Universität Bremen*

Abstract

In this paper we formally specify a large subset of Java Virtual Machine instructions based on the descriptions in the Java Virtual Machine Specification by Lindholm and Yellin, in the Java Specification by Gosling, Joy and Steele, and based on the behaviors of some test programs on Sun's implementation of the Java Virtual Machine. The formal specification describes the runtime behaviors of the instructions in related memory areas as (runtime) state transitions and most structural constraints on instructions as a compile-time (or link-time) type inference system. The latter part corresponds to a core of the Bytecode Verifier and resembles dataflow analysis and abstract interpretation. We prove properties based on the formal specification. In particular, we prove that if the type inference system can derive certain compile-time (or link-time) types for a program, then the runtime data of the program will be type-correct with respect to these types in a certain sense. Indeed, our formal specification clarifies some ambiguities and incompleteness and removes some (in our view) unnecessary restrictions in the description of the (informal) Java Virtual Machine Specification.

1 Introduction

The Java Virtual Machine (short: JVM) is a platform-independent abstract computing machine containing an instruction set and running on various memory areas. The JVM is used as an intermediate language in the implementation of the programming language `Java` and of several other languages and software systems as well.

JVM programs are written in a special file format, called `class` *file format*. A `class` file contains *JVM instructions* (short: *instructions*) and other ancillary information. An *instruction* (or a *bytecode*) consists of a one-byte *opcode* representing the operation to be performed, followed by zero or more *operands* supplying arguments or data that are used by the operation. For example, the instruction `aload index` has `aload` as the opcode and `index` as one operand.

JVM programs, or more precisely `class` files, are applications that are allowed to be used directly over the Web. Strong format and structural constraints should be imposed on `class` files to guarantee that the applications do not have security problems.

In this paper we formally specify a large subset of instructions. The specification describes the runtime behaviors of the instructions in related memory areas and give many structural constraints. The formal specification is based on the JVM Specification by Lindholm and Yellin [8] (hereafter *the JVM Specification*), some test programs on Sun's implementation of the JVM written by us and sometimes on the Java Specification by Gosling, Joy and Steele [7]. The formal specification provides a foundation for exposing the behaviors, proving properties of instructions and ensures the security of internet programming.

This paper does not formalize the complete JVM Specification. Instead, it considers only the following essential instructions: the load and store instructions for objects and integers, one

*Address: FB3 Informatik, Universität Bremen, Postfach 330440, D-28334 Bremen, Germany.
E-mail: `qian@informatik.uni-bremen.de`

arithmetic instruction for integers, the object creation instruction, several operand stack management instructions, several control transfer instructions, all method invocation instructions, the return instructions that return an object, an integer and no value, the throwing instruction and the exception handling, and the verb+jsr+ and `ret` instructions for implementing `finally`-clauses. The subset of instructions we will consider is large enough to write some real examples.

The approach of the current paper is as follows:

- We formalize an operational semantics of the instructions as a (runtime) state transition machine by defining each instruction as a state transition.

- At the same time we formulate a (compile- or link-time) type inference system, with which we can check at compile or link time the types of the runtime data at all addresses of an arbitrary JVM program. The type inference system corresponds to a core of the Bytecode Verifier. It characterizes aspects of the dataflow analysis (see e.g. [1]) and can be regarded as a description of an abstract interpretation [3, 4] (see also [12]).

- The state transition machine and type inference system build our formal specification. The state transition machine is defined only for programs, where the type inference system can successfully derive types for all runtime data. This coincides with what the JVM Specification requires (§ 6.1, page 151 in [8]).

- We finally prove some properties based on the formal specification. In particular, we prove that if the type inference system can derive certain types for a program, then the runtime data will be type-correct with respect to these types in a certain sense.

To a large extent, our formal specification is a rigorous formalization of the corresponding parts of the JVM Specification. However, there are some problems with the JVM Specification. Thus we need to extend or change the JVM Specification sometimes. Three examples are as follows:

- The JVM Specification says that all possible types of an operand stack entry or a local variable should be expressible by one type. The problem is, however, that sometimes there is no one single superinterface of two interfaces due to the presence of multiple inheritance in the subinterface relation. Our solution is to use a set of types instead of a single type to denote all possible types of an operand stack entry or a local variable. Indeed, our test JVM programs have shown that the current Sun's implementation of Bytecode Verifier considers sets of interfaces.

- The JVM Specification (page 132) uses a special type indicating an uninitialized object. We introduce two new types indicating two stages of object initialization: one says that the object is newly created by `new`; the other says that the object is currently being initialized by an instance initialization method. We distinguish between these two stages because they allow different instructions.

- The JVM Specification introduces the type `returnAddress` indicating that a value is an address and requires that a subroutine can only return to a value with the type `returnAddress`. We replace `returnAddress` by a family of `jsr` address types that are each indexed by the starting address of a subroutine. With the indices we can easily formalize the structural constraints on the `jsr` and `ret` instructions.

As mentioned, there are many things we cannot consider in the current paper:

- We do not consider multi-threads.

- We do not consider arrays and primitive types except type *int*. The latter implies that we need not consider two-word wide data. We do not consider class instantiation method `<clinit>`, access control (i.e. private, public, protected or package) modifiers, and native methods. We do not consider the instructions `lookupswitch`, `tableswitch` and `wide`.
- The JVM Specification describes quite precisely what kind of runtime exceptions can be thown during the execution of an instruction. We do not consider these exceptions in this paper.
- We do not consider the memory organization in the heap. We assume that the overflow and underflow of the operand stack have been checked elsewhere.
- The paper does not specify the `class` file format in details. We formalize only those details that are necessary for the formal specification as predefined functions. We do not consider the constant pool resolution in details. Instead, we assume that there are certain predefined functions that do the jobs properly. We assume that all classes have been loaded by a single class loader in a proper way.
- In general, a compile-time type check in JVM may need to be delayed until link time. However, we do not consider the difference between the terms "compile time" and "link time" in this paper. For simplicity, we use the term "compile time" for both.

In this paper we use $\overline{o_n}$ to denote the list of n syntactical objects $o_1, \cdots, o_n$.

2 Related work

Our work is directly related to the work on building a formal model of a subset of the JVM, called *defensive JVM* (short: *dJVM*) [2]. However, the dJVM and our work are different in at least two aspects. The main difference is that dJVM includes runtime checks to assure type-safe execution, where we design a compile-time type inference system, which assures that well-typed programs do not have runtime type errors. In addition, the current dJVM and our work consider different subsets of instructions. In particular, the current dJVM does not consider subroutines (i.e. the `jsr` and `ret` instructions), whereas we do.

As mentioned before, our formal specification considers only one single class loader. Saraswat [10] studied the problem with static type-(un)safety in `Java` in the presence of more than one class loader. He suggested two solutions to sovle the problem: one is to impose some structural constraints on class loaders; the other is to associate each type with a class loader as a tag in the JVM.

Dean [5] studied a formal model relating the static typing and dynamic linking and proved the safety of the dynamic linking with respect to the static typing. As mentioned before, our formal specification does not consider the issue between static typing and dynamic linking.

Other related work includes formalizations of `Java` [6, 9, 11].

3 JVM programs, data areas and frames

A `class` file may contain *methods*. The representation of a method contains information about the method. For example, the representation contains (the reference to) its name and its descriptor, the length of the method code, the number of local variables and the maximal length of the operand stack. The code of a method is an array of bytes representing a sequence of JVM instructions. In general, a JVM program may contain more than one class. In this paper we define a *(JVM) program* $\mathcal{P}$ as a set of *methods* $\{m_1, \ldots, m_n\}$, which may stem from different `class` files. We consider an arbitrary but fixed program $\mathcal{P}$ from now on.

A method is always associated with an *exception table* for implementing `catch` clauses in `Java`. The details are introduced in Section 8.8.

We assume that each method m_i has a unique *method reference*, denoted by m_i itself. An *address* is a pair (m, n) of a method reference m and an byte number n in the corresponding method code. For any address (r, n) and any integer m, we define $(r, n) + m := (r, n + m)$. Since we do not consider multi-threads in this paper, we assume that there is just one *program count register pc*, which contains the address of the instruction currently being executed. We use $PC(\mathcal{P})$ to denote the set of the addresses of all instructions in the program $\mathcal{P}$.

Each class or interface has a *constant pool* containing several kinds of constants, ranging from numeric literals known at compile time to symbolic references of classes, interfaces, methods and fields that must be resolved at run time. References are always initially symbolic. The constant pool for a class or interface is created when a `class` file for the class or interface is successfully loaded by the JVM. Since each instruction is stored in a `class` file, each instruction has a current constant pool.

The JVM has a *heap*, from which memory for all objects (i.e. class instances) is allocated. Heap storage for objects is reclaimed by an automatic storage management system, which we do not consider in our specification. We use $\mathcal{H}$ to range over the states of the heap. Formally, we denote a state of the heap as a mapping of object references to records, i.e. of the form $\{\cdots, objref \mapsto < \cdots, fldName = val, \cdots >, \cdots\}$, where $objref$ is an object reference, $< \cdots, fldName = val, \cdots >$ is a record, which consists of zero or more pairs of labels $fldName$ and values val.

On each method invocation, the JVM creates an array of words known as *local variables*. (A word is an abstract notion in the JVM and should have a platform-specific size.) Local variables are always one-word wide and addressed by word offsets from the base of that array. Two local variables are reserved for one *long* or *double* value and addressed by the index of the first of the variables. A *state* of n local variables is a mapping of the form $\{i \mapsto val_i \mid 0 \leq i < n\}$. We use $\mathcal{L}$ to range over states of all local variables of a particular method and $\mathcal{L}[k \mapsto val]$ the mapping that is euqal to $\mathcal{L}$ except it maps k to val.

On each method invocation, the JVM creates an *operand stack*. An operand stack containing n entries can be written as a list of the form $[val_0, \cdots, val_{n-1}]$. We use $\mathcal{S}$ to range over all operand stacks, and $\mathcal{S} + val$ the result of pushing a value val onto $\mathcal{S}$. Each entry on the operand stack is one-word wide. Values of types *long* or *double* are pushed onto the operand stack as two words.

The local variables and the operand stack created on each method invocation are allocated in a *JVM frame*. A new frame is created each time a method is invoked and destroyed when its method completes. A frame keeps the current state of the local variables and the operand stack. In addition, a frame contains other useful data and is responsible for performing dynamic linking, returning values for methods and dispatching exceptions. For simplicity we just explicitly write the current *pc*, $\mathcal{L}$, $\mathcal{S}$ and $\mathcal{H}$ in a frame.

When a new frame is created on a method invocation we need to store the old frame somewhere during the execution of the method invocation. Each JVM thread has a stack, called a *Java stack*, for doing this. Since we do not consider multi-threads in this paper, we need only to consider one Java stack. We use $\mathcal{J}$ to denote a state of the Java stack. For simplicity we assume that the Java stack stores tuples $(pc, \mathcal{L}, \mathcal{S}, rpc)$ consisting of the current frame components *pc*, $\mathcal{L}$ and $\mathcal{S}$, and the return address pc', which points to the instruction to be executed when a method invocation is finished.

For specifying the exception throwing and handing, we still need a bit component, denoted as *bc*, to record whether an exception has been thrown during the execution of an instruction.

Class name	cls	$=$	$\cdots$ a class name $\cdots$
Interface name	$inter$		$\cdots$ an interface name $\cdots$
Type $Null$			$Null$
Reference type	ref	$=$	$cls \mid inter \mid Null$
Primitive type	$prim$	$=$	$byte \mid short \mid int \mid char$ $\mid long \mid float \mid double$
Type	ty	$=$	$ref \mid prim$
Reference type set	$refs$	$=$	$\{\overline{ref_n}\}$ $(n > 0)$
Type set	tys	$=$	$refs \mid prim$
`jsr` address type			$ad(pc)$
`jsr` address type sets	ads	$=$	$\{\overline{ad(pc_n)}\}$ $(n > 0)$
Void type			$void$
Uninitialized object type	raw	$=$	$unin(pc, cls) \mid init(pc, cls)$
Unusable value type			$unusable$

Figure 1: Types and type sets

The component bc can only hold one of the two states $throw$ and $notThrow$.

A *runtime state* is a tuple of the form $\Xi = (bc, pc, \mathcal{J}, \mathcal{L}, \mathcal{S}, \mathcal{H})$. The execution of a program is specified in the current paper as a set of (runtime) state transitions.

4 Data types

Data types are either JVM types or auxiliary data types to be used for the specification. The syntax in Figure 1 includes all types of JVM and some auxiliary data types. More auxiliary data types are given in Section 6. In the rest of this paper, we will use nonterminals in the syntax, possibly with subsripts, directly as variables ranging over syntactical objects satisfying the syntax. As two examples, a variable ref_1 ranges over class names, interface names and type $Null$, and a variable $prim$ ranges over primitive types.

Class names cls and interface names $inter$ are as in `Java`. Primitive types $prim$ are those of `Java` except the `boolean` type. A reference type ref is either a class name, an interface name or the special type $Null$. Note that we do not consider arrays in this paper. The $Null$ type is used in the specification to denote the type of $null$. It belongs neither to `Java` nor to JVM. A type ty is then either a reference type or a primitive type.

Class and reference type hierarchies in JVM are as in `Java`. A class hierarchy must be a tree with the `Object` as the root. This means that every class except `Object` has exactly one direct superclass and two classes always have a least common superclass. A reference type hierarchy is a class hierarchy, where a class can in addition be a subtype of (i.e. *implement*) zero or more additional interfaces and an interface a subinterface (i.e. *extend*) zero or more other interfaces. Obviously, a reference type hierarchy may have multiple inheritance of interfaces.

Due to the existence of multiple inheritance of interfaces, it may happen that two reference types have several supertypes but none of them is the least one. In order to cope with this kind of situations, we use a notion of type sets. The JVM Specification does not mention type sets. But we know no ways to avoid them in our formal specification.

Formally, a reference type set $refs$ is a nonempty set of reference types. Intuitively, if the semantics $[\![ref]\!]$ of a reference type ref is a set (of instances), then the semantics of a reference type set $\{\overline{ref_n}\}$ is the union $\bigcup_{1\le i\le n} [\![ref_i]\!]$. A singleton reference type set $\{ref\}$ is always

Type or void	$tyOrVoid$	$=$	$ty \mid \texttt{void}$
Reference type set or uninitialized object type	$refsOrRaw$	$=$	$refs \mid raw$
Reference type set, uninitialized object type or `jsr` address type	$refsOrRawOrAds$	$=$	$refs \mid raw \mid ads$
Usable value types	$usable$	$=$	$tys \mid raw \mid ads$
Anything	any	$=$	$usable \mid unusable$

Figure 2: Auxiliary types

regarded to be equal to the element ref. A type set tys is a reference type set or a primitive type. We do not consider sets of primitive types.

In order to deal with some special values in JVM, we still need some auxiliary types in the specification of JVM, which have no counterparts in `Java`. The first kind of auxiliary types are the so-called `jsr` address types. Roughly speaking, a `jsr` address type $ad(pc)$ is the type of the addresses of all `jsr` instructions that call the subroutine with the starting address pc. Thus two types $ad(pc)$ and $ad(pc')$ with $pc \not\equiv pc'$ are regarded as distinct A `jsr` address type set ads is a nonempty set $\{\overline{ad(pc_n)}\}$ of `jsr` address types. A singleton `jsr` address type set $\{ad(pc)\}$ is always regarded to be equal to the element $ad(pc)$.

The second kind of auxiliary types consists of the $void$ type corresponding to `void` in `Java`.

The third and fourth kinds of auxiliary types are types of the forms $unin(pc, cls)$ and $init(pc, cls)$, respectively, for an address pc and a class name cls. The two kinds of types denote two degrees of initialization of objects. We use raw to range over them. $unin(pc, cls)$ and $unin(pc', cls')$ with $pc \not\equiv pc'$ or $cls \not\equiv cls'$ are two distinct types. The same holds for $init(pc, cls)$ and $init(pc', cls')$. Note that the type set $refs$ indicates that the objects have been initialized.

Finally we need the type $unusable$ indicating that the value of a local variable is unusable. Intuitively, if at a certain position of a program a local variable may hold different values of incompatible types or no values at all, then the type of the local variable is $unusable$ at this position. The type $unusable$ can be regarded as the union of all (and thus some incompatible) types.

The JVM Specification has type $returnAddress$ corresponding to the family of types $ad(pc)$, but no explicit types corresponding to $void$, raw (i.e. $unin(pc, cls)$ and $init(pc, cls)$) and $unusable$. In our paper, we use $returnAddress$ to tag runtime values in establishing relationship between runtime values and static types.

In order to enable a simple and precise description in our specification we define more auxiliary syntactical types in Figure 2.

We define the relation $\sqsubseteq$ by

$$any \sqsubseteq any' \text{ holds if and only if } any \subseteq any' \text{ holds or } any' \equiv unusable,$$

where a signle type ty is regarded as a singleton type set $\{ty\}$ in checking that $any \subseteq any'$. The intuitive meaning is that if $any \sqsubseteq any'$, then any instruction that is applicable to all values of the type any' must be applicable to all values of the type any.

The relation $\sqsubseteq$ is in fact quite subtle due to the syntax in Figures 1 and 2. For examples, if any is a $prim$ or a raw, then any' must either be $unusable$ or identical to any.

The relation $\sqsubseteq$ has nothing to do with a reference type hierarchy. Intuitively, considering the union semsntics of a type set, we could consider a reference type hierarchy in the definition of the relation $\sqsubseteq$ so that if in a type set a reference type is a subtype of another reference type, then the subtype could be removed. But we prefer not to do this, since this would need an equivalence relation among type sets and make the specification more complex.

5 Some auxiliary functions

The JVM Specification requires that a symbolic reference in a constant pool to a class, an interface, a method or a field must be resolved in certain ways. The current paper does not consider the resolution processes in details. Instead, we assume the existence of the following functions, which yield the necessary information for a given instruction via the the constant pool of the class or interface containing the instruction.

The function $cls\_Resol(index1, index2)$ yields (the reference to) the class name cls stored at the index $(index1 << 8) \mid index2$ in the constant pool.

The function $fld\_Resol(index1, index2)$ yields a tuple $(fldName, cls, ty)$ consisting of (the references to) the field name $fldName$, the name cls of the class containing the declaration of the field, and the type ty of the field, where ty cannot be $Null$. All this inforamtion is stored at the index $(index1 << 8) \mid index2$ in the constant pool.

The function $methResolSp(index1, index2)$ yields a tuple $(methName, cls, (\overline{ty_n})tyOrVoid, code, max\_locals)$ consisting of (the references to) the method name $methName$, the name cls of the class containing the declaration of the method, the descriptor $(\overline{ty_n})tyOrVoid$ of the method, the address $code$ of the code of the method and the number max_locals of local variables in the method, based on the information stored at the index $(index1 << 8) \mid index2$ in the constant pool.

The function $methResolSt(index1, index2)$ yields the same result as $methResolSp(index1, index2)$, but has a different impelementation. Although the differences are not considered in the current paper, we prefer to use different functions to avoid confusion.

The function $methResolV(index1, index2)$ yields a tuple $(methName, cls, (\overline{ty_n})tyOrVoid, entry\_index)$, where $methName$, cls and $(\overline{ty_n})tyOrVoid$ are the same as in the function $methResolSp(index1, index2)$ and $entry\_index$ is an entry index, which can be used to find the method, when a method table is given. The function $methResolV$ is used in the situation where the method table has to be determined separately based on an object. We define the function $methSelV(objref, entry\_index)$, which combines the processes of finding a method table based on an object (reference) $objref$ and of accessing the method at the index $entry\_index$ in the table, and which yields a pair $(code, max\_locals)$ consisting of the address $code$ of the code of the method and the number max_locals of local variables in the method.

The function $methResolI(index1, index2)$ yields a tuple $(methName, inter, (\overline{ty_n})tyOrVoid)$ consisting of (the references to) the method name $methName$, the name $inter$ of the interface containing the declaration of the method and the descriptor $(\overline{ty_n})tyOrVoid$ of the method, based on the information stored at the index $(index1 << 8) \mid index2$ in the constant pool. The function $methResolI$ does not yield the address of the code of the method or the entry index to the code. Therefore we define a function $methSelI(objref, mathName, (\overline{ty_n})tyOrVoid)$, which searches for a method with the name $mathName$ and the identical descriptor $(\overline{ty_n})tyOrVoid$ in a method table determined by $objref$ and yields a pair $(code, max\_locals)$ consisting of the address $code$ of the code of the method and the number max_locals of local variables in the method.

The function $offset(branch1, branch2)$ yields the address offset $(branch1 << 8) \mid branch2$.

The boolean function $inScope(pc, pc+n)$ yields false when the control transfer is not allowed to go from the program count pc to the program count $pc + n$. Concretely, for an address pc of an instruction we define that $inScope(pc, pc + n)$ yields true if and only if both following conditions are true:

- pc and $pc + n$ are in the same method code.
- $pc + n$ points to the beginning (i.e. to the opcode) of an instruction.

If we know the reference to a method code, then we can define functions that take the reference as argument and yield as results the class cls that contains the declaration of the method, the method descriptor $(\overline{ty_n})tyOrVoid$, the maximal length of the operand stack max_stack, the number of local variables max_locals and the exception handlers table of the method. For notational simplicity, however, we formally let the functions take the starting address $aMeth$ of the code of a method as argument.

- The function $methInfo(aMeth)$ yields $(cls, (\overline{ty_n})tyOrVoid, max\_locals)$.
- The function $Catch(aMeth)$ yields the list of `catch` clauses. The details will be given in Section 8.8.

6 Runtime state types

In general, there is no guarantee that any `class` file it is asked to load is properly formed. Thus the JVM Specification requires that a `class` file verifier must be used to check some constraints on the `class` file. A central part of the `class` file verifier is the so-called *Bytecode Verifier*.

The main task of Bytecode Verifier is to derive at compile time the types of the possible runtime data mainly in the local variables and in the operand stack and to ensure that no instructions will use the runtime data with wrong types.

For this purpose, we define a *local variable type table* $\mathcal{LT}$ as a mapping of the form

$$\{i \mapsto any_i \mid 0 \leq i < max\_locals\}$$

where max_locals is the number of local variables in the method. We use $\mathcal{LT}[kmapstoany]$ to denote the mapping that is euqal to $\mathcal{LT}$ except it maps k to any. Furthermore, we define an *operand type stack* $\mathcal{ST}$ as a list of the form

$$[usable_0, \cdots, usable_{top\_index}]$$

where max_stack is the maximal length of the operand stack in the method and $top\_index < max\_stack$ holds.

The syntax for $\mathcal{LT}$ and $\mathcal{ST}$ above implies that a local variable and an operand stack entry may hold values of different types. For an operand stack entry these types must be compatible, in the sense they are always denotable as the type *usable*. For a local variable these types may be incompatible, in this case they may be the type *unusable*. This difference is required by the JVM Specification. For, a local variable need not be really used in the execution, whereas an operand stack entry is normally used.

Intuitively, in order to ensure that no instructions use the runtime data with wrong types at run time, we need to ensure at compile time that if an instruction uses a local variable or an operand stack entry, then it must be able to deal with all possible types for that local variable or operand stack entry.

We define a *(runtime) state type* Π as a tuple $(\mathcal{LT}, \mathcal{ST}, \mathcal{G}, \mathcal{B}, \mathcal{U})$, where $\mathcal{G}$, $\mathcal{B}$ and $\mathcal{U}$ are auxiliary components and will be defined in Section 8.9.

The relation $\sqsubseteq$ is naturally extended as follows:

- $\mathcal{LT} \sqsubseteq \mathcal{LT}'$ holds if and only if $\mathcal{LT} \equiv \{\overline{n \mapsto any_n}\}$, $\mathcal{LT}' \equiv \{\overline{n \mapsto any'_n}\}$ and $any_i \sqsubseteq any'_i$ hold for all $i = 1, \ldots, n$.

- $\mathcal{ST} \sqsubseteq \mathcal{ST}'$ holds if and only if $\mathcal{ST} \equiv [\overline{usable_n}]$, $\mathcal{ST}' \equiv [\overline{usable'_n}]$ and $usable_i \sqsubseteq usable'_i$ hold for all $i = 1, \ldots, n$.
- $(\mathcal{LT},\mathcal{ST},\mathcal{G},\mathcal{B},\mathcal{U}) \sqsubseteq (\mathcal{LT}',\mathcal{ST}',\mathcal{G}',\mathcal{B}',\mathcal{U}')$ holds if and only if $\mathcal{LT} \sqsubseteq \mathcal{LT}'$, $\mathcal{ST} \sqsubseteq \mathcal{ST}'$, $\mathcal{G} \sqsubseteq \mathcal{G}'$, $\mathcal{B} \sqsubseteq \mathcal{B}'$ and $\mathcal{U} \sqsubseteq \mathcal{U}'$ hold, where the relations $\mathcal{G} \sqsubseteq \mathcal{G}'$, $\mathcal{B} \sqsubseteq \mathcal{B}'$ and $\mathcal{U} \sqsubseteq \mathcal{U}'$ will be given in Section 8.9.

The intuitive meaning of $\Pi \sqsubseteq \Pi'$ is that any instruction that is applicable to all states of the state type Π' must be applicable to all states of the state type Π.

The relation $\sqsubseteq$ is extremely useful in our specification. But we sometimes need more relations on the types.

To formulate a relation extending the *widening reference conversions* for non-array types (§ 5.1.4 in [7] or § 2.6.4 in [8]), we define the relation $widRefConvertible$ as the smallest reflexive and transitive relation on all reference type sets $refs$ satisfying:

$$\begin{array}{rl}
widRefConvertible(cls, cls') \text{ holds} & \text{if } cls \text{ extends } cls' \\
widRefConvertible(cls, inter) \text{ holds} & \text{if } cls \text{ implements } inter \\
widRefConvertible(inter, inter') \text{ holds} & \text{if } inter \text{ extends } inter' \\
widRefConvertible(inter, Object) \text{ holds} & \\
widRefConvertible(Null, ref) \text{ holds} & \\
widRefConvertible(\{\overline{ref_n}\}, \{\overline{ref'_m}\}) \text{ holds} & \text{if } \forall(1 \le i \le n).\exists(1 \le j \le m). \\
& \quad widRefConvertible(ref_i, ref'_j)
\end{array}$$

A special case of the relation $widRefConvertible$ is the relation $isDirectSubcls$ on the classes defined by

$$isDirectSubcls(cls', cls) \;\text{ if and only if }\; cls \text{ extends } cls'$$

To formulate the *widening primitive conversions* (see § 5.1.2 in [7] or § 2.6.2 in [8]), we define the relation $widPrimConvertible$ as the smallest reflexive and transitive relation on all primitive types containing

$$\{(byte, short), (short, int), (char, int), (int, long), (long, float), (float, double)\}$$

To constrain the types of the actual and formal parameters in a method invocation we define the relation $invoConvertible$ on all type sets as

$$invoConvertible := widRefConvertible \cup widPrimConvertible.$$

To constrain the types of the variable and the value in an assignment, we define the relation $assConvertible$ on all type sets as

$$assConvertible := invoConvertible.$$

Note that the Java Specification or JVM Specification requires that the relation $assConvertible$ extends the relation $invoConvertible$ by some narrowing primitive conversions for integer constants. We do not consider this difference in the current paper for simplicity.

7 The judgements and the forms of rules

This paper deals with two kinds of judgements. The first kind consists of (runtime) state transitions of the form $\Xi \implies \Xi'$, meaning that an instruction transforms a runtime state Ξ into another runtime state Ξ'. The second kind of judgements is of the form $pc \vdash \Pi$, meaning that all possible runtime states at an address pc have the compile-time state type Π. Sometimes the state type at an address depends not only on the instruction at the address but also on other instructions in the program through control flow. Thus we also use judgements of the form $pc \vdash_{\sqsubseteq} \Pi$ to specify a part of the state type at pc, meaning that every state type Π' at pc must satisfy that $\Pi \sqsubseteq \Pi'$.

Our formal specification consists of two categories of descriptions. The first defines (runtime) state transitions. The data in a runtime state are untyped except that their lengths (i.e. whether they are a byte, one-word or double-word wide) are known. The definitions of state transitions are of the following form:

$$\frac{\cdots\ conditions\ \cdots}{\Xi[\overline{A_n}] \implies \Xi[\overline{B_n}]}$$

It means that if the *conditions* hold, then any state $\Xi[\overline{A_n}]$ will be transformed into $\Xi[\overline{B_n}]$, where $\Xi[\overline{A_n}]$ denotes a state containing the components $\overline{A_n}$ and $\Xi[\overline{B_n}]$ is the state obtained from $\Xi[\overline{A_n}]$ by replacing each A_i by B_i for $i = 1, \ldots, n$.

The second category defines a compile-time type inference system. The purpose of the type inference system is to check and derive a state type Π for each address pc of an instruction in a program. Types and type sets in a state type follow the syntax Figures 1 and 2. The typing rules are of the form

$$\frac{\begin{array}{c}\boxed{conditions1}\\ conditions2\end{array}}{pc' \vdash_{\sqsubseteq} \Pi}$$

It means that if all conditions in $\boxed{conditions1}$ hold, then all conditions in *conditions* hold; furthermore, if all conditions in both $\boxed{conditions1}$ and *conditions*2 hold, then $pc' \vdash_{\sqsubseteq} \Pi$ holds. If two state types in one rule are of the forms $\Pi[\overline{S_n}]$ and $\Pi[\overline{T_n}]$, then one of them, say $\Pi[\overline{S_n}]$, can be obtained from the other, say $\Pi[\overline{T_n}]$, by replacing each T_i by S_i for $i = 1, \ldots, n$.

A special case of the above form is

$$\frac{\begin{array}{c}\boxed{conditions1}\\ conditions2\end{array}}{}$$

It means that if all conditions in $\boxed{conditions1}$ hold, then all conditions in *conditions*2 hold.

8 A specification of a subset of JVM instructions

It is always assumed in the specification that if the instruction at pc changes the current program count into pc', then $inScope(pc, pc')$ always holds.

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

The bit component bc in a runtime state can only be set to $notThrow$ or $throw$ and it is set to $throw$ if and only if an exception is thrown. Since the bit component bc in all runtime states in the following subsections except in Subsection 8.8 is set to $notThrow$, we omit the bit component bc in the following subsections except in Subsection 8.8.

In this section we do not consider the components $\mathcal{G}$, $\mathcal{B}$ and $\mathcal{U}$ in the state type until Section 8.9.

8.1 The `main` method

The JVM Specification requires that a JVM program starts execution by invoking the method `main` of some specified class, passing it an array of strings as the single argument. The method `main` must be declared as `public` and `static` and with return type `void`. In our specification a JVM program starts execution with the instruction `invokestatic` on the `static` method `main`, which has no return value type. For simplicity, we also assume that the method `main` do not have arguments.

We use the notation $\mathcal{L}^n$ to denote an arbitrary unknown state of n local variables. Intuitively, $\mathcal{L}^n$ represents a state of the memory before it has been allocated for local variables.

Let $\mathcal{H}_{start}$ be an initial heap state, about which no details are considered in this paper.

We formally define the following special state transition, which creates a runtime state. Remember that the bit component on the runtime state is implicitly set to *notThrow*.

$$\frac{(\texttt{main}, cls, ()Void, code, max\_locals) = methResolSt(index1, index2)}{\Longrightarrow\ (code, [], \mathcal{L}^{max\_locals}, [], \mathcal{H}_{start})}$$

The typing rule for the starting address of the method `main` is given as follows. The definitions of the notations $\mathcal{G}_{start}$, $\mathcal{B}_{start}$ and $\mathcal{U}_{start}$ will be given in Section 8.9.

$$\frac{(\texttt{main}, cls, ()Void, code, max\_locals) = methResolSt(index1, index2)}{code \vdash_{\sqsubseteq} (\{i \mapsto unusable \mid 0 \leq i < max\_locals\}, [], \mathcal{G}_{start}, \mathcal{B}_{start}, \mathcal{U}_{start})}$$

8.2 Load and store instructions

The `aload` and `iload` instructions load a local variable onto the operand stack.

$$\frac{\mathcal{P}(pc) \equiv \texttt{aload index} \text{ or } \texttt{iload index}}{\Xi[pc, \mathcal{L}, \mathcal{S}] \implies \Xi[pc+2, \mathcal{L}, \mathcal{S} + \mathcal{L}(index)]}$$

The typing rule for `aload` requires that the $\mathcal{L}(index)$ in the above state transition must be of a type $refsOrRaw$:

$$\frac{\boxed{\mathcal{P}(pc) \equiv \texttt{aload index} \qquad pc \vdash \Pi[\mathcal{LT}, \mathcal{ST}]} \quad refsOrRaw = \mathcal{LT}(index)}{pc + 2 \vdash_{\sqsubseteq} \Pi[\mathcal{LT}, \mathcal{ST} + refsOrRaw]}$$

The typing rule for `iload` requires that the $\mathcal{L}(index)$ is int.

$$\frac{\boxed{\mathcal{P}(pc) = \texttt{iload index} \qquad pc \vdash \Pi[\mathcal{LT}, \mathcal{ST}]} \quad int = \mathcal{LT}(index)}{pc + 2 \vdash_{\sqsubseteq} \Pi[\mathcal{LT}, \mathcal{ST} + int]}$$

The `astore` and `istore` instructions store a value from the operand stack into a local variable.

$$\frac{\mathcal{P}(pc) \equiv \texttt{astore index} \text{ or } \texttt{istore index}}{\Xi[pc, \mathcal{L}, \mathcal{S} + val] \implies \Xi[pc+2, \mathcal{L}[index \mapsto val], \mathcal{S}]}$$

The typing rule for `astore` requires that the val in the above is a $refsOrRawOrAds$:

$$\frac{\boxed{\mathcal{P}(pc) \equiv \texttt{astore index}} \quad pc \vdash \Pi[\mathcal{LT}, \mathcal{ST} + refsOrRawOrAds]}{pc + 2 \vdash_{\sqsubseteq} \Pi[\mathcal{LT}[index \mapsto refsOrRawOrAds], \mathcal{ST}]}$$

Note that **astore** may store an address, whereas **aload** cannot load an address.

The typing rule for **istore** is as follows:

$$\frac{\boxed{\mathcal{P}(pc) = \texttt{istore index}} \quad pc \vdash \Pi[\mathcal{LT}, \mathcal{ST} + int]}{pc + 2 \vdash_{\sqsubseteq} \Pi[\mathcal{LT}[index \mapsto int], \mathcal{ST}]}$$

The **aconst_null** instruction loads the *null* reference onto the operand stack.

$$\frac{\mathcal{P}(pc) \equiv \texttt{aconst\_null}}{\Xi[pc, \mathcal{S}] \implies \Xi[pc + 1, \mathcal{S} + null]}$$

The type of *null* is *Null*. Note that *Null* is a special *ref*.

$$\frac{\boxed{\mathcal{P}(pc) \equiv \texttt{aconst\_null} \qquad pc \vdash \Pi[\mathcal{ST}]}}{pc + 1 \vdash_{\sqsubseteq} \Pi[\mathcal{ST} + Null]}$$

The **bipush** instruction loads an integer constant onto the operand stack.

$$\frac{\mathcal{P}(pc) \equiv \texttt{bipush byte}}{\Xi[pc, \mathcal{S}] \implies \Xi[pc + 2, \mathcal{S} + int\_extension(byte)]}$$

where the function *int_extension* extends a *byte* into a (one-word wide) integer.

The typing rule for **bipush** is as follows:

$$\frac{\boxed{\mathcal{P}(pc) \equiv \texttt{bipush byte} \qquad pc \vdash \Pi[\mathcal{ST}]}}{pc + 2 \vdash_{\sqsubseteq} \Pi[\mathcal{ST} + int]}$$

The **getfield** instruction replaces the object reference *objref* at the top of the operand stack by the value *val* of a field of the object. Note that the value *val* can be a reference itself or a value of a primitive type.

$$\frac{\begin{array}{l}\mathcal{P}(pc) \equiv \texttt{getfield index1 index2} \\ (fldName, \_, ty) = fld\_Resol(index1, index2) \\ \mathcal{H}(objref) = \langle \cdots, fldName = val, \cdots \rangle\end{array}}{\Xi[pc, \mathcal{S} + objref, \mathcal{H}] \implies \Xi[pc + 3, \mathcal{S} + val, \mathcal{H}]}$$

The typing rule cannot guarantee that *objref* really exists in $\mathcal{H}$ at run time, since *Null* is a special *ref*. If *objref* does not exist in $\mathcal{H}$, then there will be a runtime exception.

$$\frac{\begin{array}{l}\boxed{\mathcal{P}(pc) \equiv \texttt{getfield index1 index2}} \\ pc \vdash \Pi[\mathcal{ST} + refs] \\ (\_, cls, ty) = fld\_Resol(index1, index2) \qquad widRefConvertible(refs, cls)\end{array}}{pc + 3 \vdash_{\sqsubseteq} \Pi[\mathcal{ST} + ty]}$$

Note that *ty* must be either a class, an interface or a primitive type.

The condition $widRefConvertible(refs, cls)$ appears to be a quite natural condition in the above typing rule. The JVM Specification does not require such a condition for the **getfield** instruction.

The `putfield` instruction stores the value at the top of the operand stack into a field of the object referenced by the reference at the second top of the operand stack.

$$\frac{\begin{array}{l}\mathcal{P}(pc) \equiv \texttt{putfield index1 index2} \\ (fldName, \_, ty) = fld\_Resol(index1, index2) \\ \mathcal{H}(objref) = \langle \cdots, fldName = \_, \cdots \rangle\end{array}}{\Xi[pc, \mathcal{S} + objref + val, \mathcal{H}] \implies \Xi[pc+3, \mathcal{S}, \mathcal{H}[objref \mapsto \langle \cdots, fldName = val, \cdots \rangle]]}$$

The JVM Specification (page 326) requires that the type of the val stored by a `putfield` instruction must be compatible with the descriptor of the field of the class instance being stored into. This requirement is, strictly speaking, a dynamic type checking, since "the type of val" can only be known at run time. The following typing rule will specify the compile-time type checking using the statically derived type for val. In Section 10 we will introduce a type tag for each value and prove that the compile-time type-correctness in our specification ensures the type-correctness at run time.

$$\frac{\begin{array}{l}\boxed{\mathcal{P}(pc) \equiv \texttt{putfield index1 index2}} \\ pc \vdash \Pi[\mathcal{ST} + refs + tys] \\ (\_, cls, ty) = fld\_Resol(index1, index2) \\ widRefConvertible(refs, cls) \qquad assConvertible(tys, ty)\end{array}}{pc+3 \vdash_{\sqsubseteq} \Pi[\mathcal{ST}]}$$

Note that $widRefConvertible$ is a relation on reference type sets, whereas $assConvertible$ is a relation on all type sets.

8.3 Arithmetic instructions

We only consider the arithmetic instruction `iadd` in this paper. Other arithmetic instructions can be dealt with in a very similar way.

$$\frac{\mathcal{P}(pc) = \texttt{iadd}}{\Xi[pc, \mathcal{S} + val1 + val2] \implies \Xi[pc+1, \mathcal{S} + (val1 + val2)]}$$

Note that in $\mathcal{S} + (val + val)$ the first $+$ denotes the push operation on the operand stack and the second $+$ denotes the integer plus operation.

The typing rule for `iadd` is simple:

$$\frac{\begin{array}{c}\boxed{\mathcal{P}(pc) = \texttt{iadd}} \\ pc \vdash \Pi[\mathcal{ST} + int + int]\end{array}}{pc+3 \vdash_{\sqsubseteq} \Pi[\mathcal{ST} + int]}$$

8.4 Object creation

A `new` instruction creates an object $\langle \cdots \rangle$ with the default values for its instance variables.

$$\frac{\begin{array}{l}\mathcal{P}(pc) \equiv \texttt{new index1 index2} \\ cls = cls\_Resol(index1, index2) \\ objref \text{ is new in } \mathcal{H} \qquad \langle \cdots \rangle \text{ is a new object of the class } cls\end{array}}{\Xi[pc, \mathcal{S}, \mathcal{H}] \implies \Xi[pc+3, \mathcal{S} + objref, \mathcal{H} \cup \{objref \mapsto \langle \cdots \rangle\}]}$$

The JVM Specification requires informally (page 132) that "the (Bytecode) Verifier rejects code that uses the new object before it has been initialized or that initializes the object twice". But the use of the word "use" in a specification is problematic, since different instructions may "use" an object in different ways. Although the JVM Specification sometimes replaces the word "use" by "access or modify", the problem remains. For example, test programs show that Sun's implementation allows "use" of a reference to a new object in an `aload` and `astore`, instruction, but not in a `if_acmpeq` instruction. The JVM Specification does not specify this.

In this paper we still use the word "use" in informal explanations. But we do introduce a special type in our formal specification to indicate that the object is still uninitialized[1]. The special type is written as $unin(pc, cls)$, where pc is the address of the `new` instruction and cls is the class, of which an object has been created. The address pc in $unin(pc, cls)$ is necessary to distinguish between the objects created for the same class by different `new` instructions. The conditions in a typing rule specify precisely how an instruction behaves on an uninitialized object.

Now we give the typing rule for `new`:

$$\frac{\boxed{\mathcal{P}(pc) \equiv \texttt{new index1 index2} \qquad pc \vdash \Pi[\mathcal{ST}]} \quad cls = cls\_Resol(index1, index2)}{pc + 3 \vdash_{\sqsubseteq} \Pi[\mathcal{ST} + unin(pc, cls)]}$$

8.5 Operand stack management instructions

We consider only three instructions for the manipulation of the operand stack. Remember that we do not consider double-word wide data in this paper. Thus the following rules implicitly assume that the top element in the operand stack is always one-word wide. The rules are quite straightforward.

$$\frac{\mathcal{P}(pc) = \texttt{dup}}{\Xi[pc, \mathcal{S} + word] \implies \Xi[pc + 1, \mathcal{S} + word + word]}$$

$$\frac{\boxed{\mathcal{P}(pc) = \texttt{dup}} \quad pc \vdash \Pi[\mathcal{ST} + usable]}{pc + 1 \vdash_{\sqsubseteq} \Pi[\mathcal{ST} + usable + usable]}$$

$$\frac{\mathcal{P}(pc) = \texttt{pop}}{\Xi[pc, \mathcal{S} + word] \implies \Xi[pc + 1, \mathcal{S}]}$$

$$\frac{\boxed{\mathcal{P}(pc) = \texttt{pop}} \quad pc \vdash \Pi[\mathcal{ST} + usable]}{pc + 1 \vdash_{\sqsubseteq} \Pi[\mathcal{ST}]}$$

$$\frac{\mathcal{P}(pc) = \texttt{swap}}{\Xi[pc, \mathcal{S} + word1 + word2] \implies \Xi[pc + 1, \mathcal{S} + word2 + word1]}$$

$$\frac{\boxed{\mathcal{P}(pc) = \texttt{swap}} \quad pc \vdash \Pi[\mathcal{ST} + usable + usable']}{pc + 1 \vdash_{\sqsubseteq} \Pi[\mathcal{ST} + usable' + usable]}$$

[1]The JVM Specification has suggested the special type. But she does not use it in the specification.

8.6 Control transfer instructions

We consider the following control transfer instructions. Other control transfer instructions can be dealt with in a very similar way.

The state transitions for the control transfer instructions are quite straightforward.

$$\frac{\mathcal{P}(pc) \equiv \texttt{if\_acmpeq branch1 branch2}}{\Xi[pc, \mathcal{S} + val1 + val2] \implies \Xi[(val1 == val2)?\ pc + \mathit{offset}(branch1, branch2) : pc + 3, \mathcal{S}]}$$

where $val1 == val2$ checks whether two references are equal, i.e. whether the references point to the same object.

$$\frac{\mathcal{P}(pc) = \texttt{if\_icmpeq branch1 branch2}}{\Xi[pc, \mathcal{S} + val1 + val2] \implies \Xi[(val1 ==_{int} val2)?\ pc + \mathit{offset}(branch1, branch2) : pc + 3, \mathcal{S}]}$$

$$\frac{\mathcal{P}(pc) = \texttt{goto branch1 branch2}}{\Xi[pc] \implies \Xi[\mathit{offset}(branch1, branch2)]}$$

The typing rules are straightforward, too. Remember that it is always implicitly checked that $inScope(pc + \mathit{offset}(branch1, branch2))$ and $inScope(pc + 3)$ hold.

$$\frac{\boxed{\mathcal{P}(pc) \equiv \texttt{if\_acmpeq branch1 branch2}} \quad pc \vdash \Pi[\mathcal{ST} + refs + refs']}{pc + \mathit{offset}(branch1, branch2) \vdash_{\sqsubseteq} \Pi[\mathcal{ST}] \quad pc + 3 \vdash_{\sqsubseteq} \Pi[\mathcal{ST}]}$$

$$\frac{\boxed{\mathcal{P}(pc) = \texttt{if\_icmpeq branch1 branch2}} \quad pc \vdash \Pi[\mathcal{ST} + int + int]}{pc + \mathit{offset}(branch1, branch2) \vdash_{\sqsubseteq} \Pi[\mathcal{ST}] \quad pc + 3 \vdash_{\sqsubseteq} \Pi[\mathcal{ST}]}$$

$$\frac{\boxed{\mathcal{P}(pc) = \texttt{goto branch1 branch2}} \quad pc \vdash \Pi}{pc + \mathit{offset}(branch1, branch2) \vdash_{\sqsubseteq} \Pi}$$

There is a tricky point in the above typing rules. The JVM Specification 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. The purpose is to prevent a devious piece of code to fool the verifier into thinking it had initialized an uninitialized object when it had, in fact, initialized an object created in a previous pass through the loop. Although this might be true in Sun's implementation, this cannot happen according to our specification. The reason is that if there is a possibility that an operand stack entry or a local variable at a position in a program contains references to both uninitialized and initialized objects, then the type *unusable* must be derivable for it (Proposition 11.1). This means that any reference in the operand stack entry or local variable is practically useless.

8.7 Method invocation and return instructions

8.7.1 State transitions for method invocation instructions

We first consider the state transition for the `invokespecial` instruction. Since the instruction is only used to invoke constructor methods and private methods and to perform method invocations via `super`, we use the function $methResolSp$ to compute the code of the method based on the information stored at the index determined by $index1$ and $index2$ in the current constant pool. The state transition says that the execution of the method should start with a runtime state, in which the bit component is set to $notThrow$, the operand stack is empty and the local variables hold the object, on which the method is invoked, together with all actual parameters. Remember that the notation $\mathcal{L}^n$ denotes an arbitrary but unknown state of n local variables.

$$\frac{\begin{array}{l}\mathcal{P}(pc) \equiv \texttt{invokespecial index1 index2}\\ (methName, cls, (\overline{ty_n})tyOrVoid, code, max\_locals) = methResolSp(index1, index2)\end{array}}{\begin{array}{l}\Xi[pc, \mathcal{J}, \mathcal{L}, \mathcal{S} + objref\overline{+val_n}]\\ \quad \Longrightarrow \Xi[code, \mathcal{J} + (pc, \mathcal{L}, \mathcal{S}, pc+3), \mathcal{L}^{max\_locals}[0 \mapsto objref, \overline{n \mapsto val_n}], []]\end{array}}$$

The state transition for `invokevirtual` is much like that for `invokespecial`. The difference is that the former uses the functions $methResolV$ and $methSelV$ to find the code of the method in the method table that is determined by $objref$, whereas the latter uses the function $methResolSp$ to find the code of the method in a method table independent of $objref$.

$$\frac{\begin{array}{l}\mathcal{P}(pc) \equiv \texttt{invokevirtual index1 index2}\\ (methName, cls, (\overline{ty_n})tyOrVoid, entry\_index) = methResolV(index1, index2)\\ methName \not\equiv \texttt{<init>}\\ (code, max\_locals) = methSelV(objref, entry\_index)\end{array}}{\begin{array}{l}\Xi[pc, \mathcal{J}, \mathcal{L}, \mathcal{S} + objref\overline{+val_n}]\\ \quad \Longrightarrow \Xi[code, \mathcal{J} + (pc, \mathcal{L}, \mathcal{S}, pc+3), \mathcal{L}^{max\_locals}[0 \mapsto objref, \overline{n \mapsto val_n}], []]\end{array}}$$

The state transition for `invokeinterface` is much like that for `invokevirtual`. The main difference is that the former uses the function $methSelI$, which searches for the method in the method table, instead of the procedure $methSelV$, which accesses the method via the entry index in the method table.

$$\frac{\begin{array}{l}\mathcal{P}(pc) \equiv \texttt{invokeinterface index1 index2 nargs 0}\\ (mathName, inter, (\overline{ty_{nargs-1}})tyOrVoid) = methResolI(index1, index2)\\ (code, max\_locals) = methSelI(objref, mathName, (\overline{ty_{nargs-1}})tyOrVoid)\end{array}}{\begin{array}{l}\Xi[pc, \mathcal{J}, \mathcal{L}, \mathcal{S} + objref\overline{+val_{nargs-1}}]\\ \quad \Longrightarrow \Xi[code, \mathcal{J} + (pc, \mathcal{L}, \mathcal{S}, pc+5), \mathcal{L}^{max\_locals}[0 \mapsto objref, \overline{nargs-1 \mapsto val_{nargs-1}}], []]\end{array}}$$

The state transition for `invokestatic` is much like that for `invokespecial`. One difference is that the former does not have an object, on which the method is invoked, whereas the latter has one. Another difference is that the former uses the function $methResolSt$, whereas the latter the function $methResolSp$.

$$\frac{\begin{array}{l}\mathcal{P}(pc) \equiv \texttt{invokestatic index1 index2}\\ (methName, cls, (\overline{ty_n})tyOrVoid, code, max\_locals) = methResolSt(index1, index2)\end{array}}{\begin{array}{l}\Xi[pc, \mathcal{J}, \mathcal{L}, \mathcal{S}\overline{+val_n}]\\ \quad \Longrightarrow \Xi[code, \mathcal{J} + (pc, \mathcal{L}, \mathcal{S}, pc+3), \mathcal{L}^{max\_locals}[i \mapsto val_{i+1} \mid 0 \leq i < n], []]\end{array}}$$

8.7.2 Types at the starting addresses of method bodies

A method invocation causes the execution of a method body. Thus we include the typing rules for the starting address of a method body in this section. However, it is important to note that these typing rules are totally independent of method invocations.

We first consider the typing rule for a `<init>` method in a class that is not $Object$. The definitions of the notations $\mathcal{G}_{start}$, $\mathcal{B}_{start}$ and $\mathcal{U}_{start}$ will be given in Section 8.9.

$$\frac{\boxed{pc \text{ is the start address of an instance initialization method } \texttt{<init>}} \\ (cls, (\overline{ty_n})void, max\_locals) = methInfo(pc) \qquad cls \not\equiv Object}{pc \vdash_{\sqsubseteq} (\{0 \mapsto init(pc, cls), \overline{n \mapsto ty_n}, i \mapsto unusable \mid n < i < max\_locals\}, [], \mathcal{G}_{start}, \mathcal{B}_{start}, \mathcal{U}_{start})}$$

Note that the types of the ith local variables with $n < i < max\_locals$ are *unusable*. This ensures that these variables cannot be used before they obtain some values.

As a subtle point, note also that in the above rule the object to be initialized has type $init(pc, cls)$ in the `<init>` method. In other words, we introduce yet another new type $init(pc, cls)$, instead of using $unin(pc, cls)$ or even cls. The reason is that we want to describe precisely that each `<init>` method, except in the class *Object*, must call exactly once another `<init>` on the object to be initialized before the object is used, and that the called `<init>` is either of the same class or of the immediate superclass. We do not use $unin(pc, cls)$ or cls, since the type $unin(pc, cls)$ is used only in the case, where an `<init>` must be of the class cls, i.e. not of the immediate superclass, and cls in the case, where an `<init>` cannot be invoked at all (see Section 8.7.3). In addition, we cannot use cls because it would mean that the use of the object at the current address is already allowed.

Note that the above rule, following the JVM Specification and the current Sun's implementation, does not forbit an `<init>` method to invoke itself, though the Java Specification (§ 8.6.5) explicitly disallows this.

The second rule deals with an `<init>` method in the class *Object*. The rule, in cooperation with other rules in Section 8.7.3, practically specifies that an `<init>` method in *Object* will not invoke an `<init>` method any more.

$$\frac{\boxed{pc \text{ is the start address of an instance initialization method } \texttt{<init>}} \\ (Object, (\overline{ty_n})void, max\_locals) = methInfo(pc)}{pc \vdash_{\sqsubseteq} (\{0 \mapsto Object, \overline{n \mapsto ty_n}, i \mapsto unusable \mid n < i < max\_locals\}, [], \mathcal{G}_{start}, \mathcal{B}_{start}, \mathcal{U}_{start})}$$

The typing rule for an instance method is given by the following rule.

$$\frac{\boxed{pc \text{ is the start address of an instance method}} \\ (cls, (\overline{ty_n})tyOrVoid, max\_locals) = methInfo(pc)}{pc \vdash_{\sqsubseteq} (\{0 \mapsto cls, \overline{n \mapsto ty_n}, i \mapsto unusable \mid n < i < max\_locals\}, [], \mathcal{G}_{start}, \mathcal{B}_{start}, \mathcal{U}_{start})}$$

The typing rule for a static method is given by the following rule.

$$\frac{\boxed{pc \text{ is the start address of a static method}} \\ (cls, (\overline{ty_n})tyOrVoid, max\_locals) = methInfo(pc)}{pc \vdash_{\sqsubseteq} (\{i \mapsto ty_{i+1} \mid 0 \le i < n, i \mapsto unusable \mid n \le i < max\_locals\}, [], \mathcal{G}_{start}, \mathcal{B}_{start}, \mathcal{U}_{start})}$$

8.7.3 Typing method invocation instructions

We have three typing rules for `invokespecial`. Remember that we do not consider private methods.

Two rules deal with the invocation of an instance instantiation method `<init>`. The first typing rule for `invokespecial` treats the invocation of an `<init>` on an uninitialized object immediately created by `new`. As shown in the typing rule for `new`, such an object has a type $unin(pc', cls)$. The rule here requires that the class cls containing the `<init>` must be exactly the class cls of the uninitialized object, and that the type $unin(pc', cls)$ of the object will become the type cls, indicating that the uninitialized object becomes initialized after the invocation.

$$\frac{\begin{array}{l}\boxed{\mathcal{P}(pc) \equiv \texttt{invokespecial index1 index2}}\\ pc \vdash \Pi[\mathcal{LT}, \mathcal{ST} + unin(pc', cls)\overline{+tys_n}]\\ (\texttt{<init>}, cls, (\overline{ty_n})void, \_) = methResolSp(index1, index2)\\ invoConvertible(tys_i, ty_i) \quad (i = 1, \ldots, n)\end{array}}{pc + 3 \vdash_{\sqsubseteq} \Pi[\mathcal{LT}[cls/unin(pc', cls)], \mathcal{ST}[cls/unin(pc', cls)]]}$$

Now we turn to consider `invokespecial` for the invocation of an `<init>` method inside an `<init>` method. The object to be initialized here must have the type $init(pc', cls')$, which is introduced by the typing rule (in Section 8.7.2) for the start of the enclosing `<init>` method. The class cls' in $init(pc', cls')$ must be the immediate superclass of or identical to the class cls of the invoked `<init>` method. After the invocation the object becomes initialized (within the enclosing `<init>` method).

$$\frac{\begin{array}{l}\boxed{\mathcal{P}(pc) \equiv \texttt{invokespecial index1 index2}}\\ pc \vdash \Pi[\mathcal{LT}, \mathcal{ST} + init(pc', cls')\overline{+tys_n}]\\ (\texttt{<init>}, cls, (\overline{ty_n})void, \_) = methResolSp(index1, index2)\\ isDirectSubcls(cls', cls) \text{ or } cls \equiv cls'\\ invoConvertible(tys_i, ty_i) \quad (i = 1, \ldots, n)\end{array}}{pc + 3 \vdash_{\sqsubseteq} \Pi[\mathcal{LT}[cls'/init(pc', cls')], \mathcal{ST}[cls'/init(pc', cls')]]}$$

The third typing rule for `invokespecial` deals with the invocation of a usual instance method (e.g. an invocation with `super`). In this case, we use the relation *widRefConvertible* to ensure that the class cls containing the declaration of the method is a superclass of all possible classes of the object. Note that the method may have a return type or not. The rule here deals with the case where the method has a return type. The rule for a method with no return type can be obtained by replacing $(\overline{ty_n})ty$ by $(\overline{ty_n})Void$ and $\Pi[\mathcal{ST} + ty]$ by $\Pi[\mathcal{ST}]$.

$$\frac{\begin{array}{l}\boxed{\mathcal{P}(pc) \equiv \texttt{invokespecial index1 index2}}\\ pc \vdash \Pi[\mathcal{ST} + refs\overline{+tys_n}]\\ (methName, cls, (\overline{ty_n})ty, \_) = methResolSp(index1, index2) \qquad methName \not\equiv \texttt{<init>}\\ widRefConvertible(refs, cls) \qquad invoConvertible(tys_i, ty_i) \quad (i = 1, \ldots, n)\end{array}}{pc + 3 \vdash_{\sqsubseteq} \Pi[\mathcal{ST} + ty]}$$

The typing rule for `invokevirtual` is similar to the last rule for `invokespecial`. The only difference is that the function *methResolV* is used, instead of *methResolSp*. The discussion on the return type for the last rule for `invokespecial` also holds here.

$$\frac{\begin{array}{l}\boxed{\mathcal{P}(pc) \equiv \texttt{invokevirtual index1 index2}}\\ pc \vdash \Pi[\mathcal{ST} + refs\overline{+tys_n}]\\ (methName, cls, (\overline{ty_n})ty, \_) = methResolV(index1, index2) \qquad methName \not\equiv \texttt{<init>}\\ widRefConvertible(refs, cls) \qquad invoConvertible(tys_i, ty_i) \quad (i = 1, \ldots, n)\end{array}}{pc + 3 \vdash_{\sqsubseteq} \Pi[\mathcal{ST} + ty]}$$

The typing rule for `invokeinterface` is also similar to the last rule for `invokespecial`. The differences are that the function *methResolI* is used, instead of *methResolSp*, and that the number *nargs* is explicitly given in the instruction. Note that the bytes `nargs` and `0` are contained in the instruction for historical reasons. The discussion on the return type for the last rule for `invokespecial` also holds here.

$$\frac{\begin{array}{l}\boxed{\mathcal{P}(pc) \equiv \texttt{invokeinterface index1 index2 nargs 0}} \\ nargs > 0 \\ pc \vdash \Pi[\mathcal{ST} + refs \overline{+ tys_{nargs-1}}] \\ (mathName, inter, (\overline{ty_{nargs-1}})tyOrVoid) = methResolI(index1, index2) \\ methName \not\equiv \texttt{<init>} \\ widRefConvertible(refs, inter) \quad invoConvertible(tys_i, ty_i) \quad (i = 1, \ldots, nargs - 1)\end{array}}{pc + 5 \vdash_{\sqsubseteq} \Pi[\mathcal{ST} + ty]}$$

Finally, we consider the typing rule for `invokestatic`. Again, the rule is similar to the last rule for `invokespecial`. Two differences are that the former does not have an object, on which the method is invoked, whereas the latter has one, and that the former uses the function *methResolSt*, whereas the latter the function *methResolSp*. The discussion on the return type for the last rule for `invokespecial` also holds here. Note that a static method cannot be named as `<init>` or `main`.

$$\frac{\begin{array}{l}\boxed{\mathcal{P}(pc) \equiv \texttt{invokestatic index1 index2}} \\ pc \vdash \Pi[\mathcal{ST} \overline{+ tys_n}] \\ (methName, cls, (\overline{ty_n})ty, \_) = methResolSt(index1, index2) \\ methName \not\equiv \texttt{<init>} \quad methName \not\equiv \texttt{main} \\ invoConvertible(tys_i, ty_i) \quad (i = 1, \ldots, n)\end{array}}{pc + 3 \vdash_{\sqsubseteq} \Pi[\mathcal{ST} + ty]}$$

The JVM Specification (e.g. page 262) requires that in a method invocation the types of the formal parameters of the method should be consistent with the types of the actual parameters on the operand stack. But it does not say what the consistency means. Our rules above adopt the approach in the Java Specification by using the relation *invoConvertible*.

8.7.4 Return instructions

The state transition for the `areturn` (or `ireturn`) instruction transfers the control back to the last invoking method and stores an object (or an integer) from the current operand stack as the return value onto the operand stack of the last invoking method. Note that the state transition uses the return address *rpc* stored in the current Java stack.

$$\frac{\mathcal{P}(pc) \equiv \texttt{areturn} \text{ or } \texttt{ireturn}}{\Xi[pc, \mathcal{J} + [(\_, \mathcal{L}', \mathcal{S}', rpc)], \mathcal{L}, \mathcal{S} + val] \implies \Xi[rpc, \mathcal{J}, \mathcal{L}', \mathcal{S}' + val]}$$

The state transition for the `return` instruction is similar, except that the instruction returns no value.

$$\frac{\mathcal{P}(pc) \equiv \texttt{return}}{\Xi[pc, \mathcal{J} + [(\_, \mathcal{L}', \mathcal{S}', retpc)], \mathcal{L}, \mathcal{S}] \implies \Xi[retpc, \mathcal{J}, \mathcal{L}', \mathcal{S}']}$$

In a method `main`, where the Java stack is empty, a `return` instruction leads to termination of the program:

$$\frac{\mathcal{P}(pc) \equiv \texttt{return}}{\Xi[pc, [], \mathcal{L}, \mathcal{S}] \implies}$$

The following rule ensures that the object that the `areturn` instruction returns has a correct type:

$$\frac{\boxed{\mathcal{P}(pc) \equiv \texttt{areturn}}}{pc \vdash \Pi[\mathcal{ST} + refs]} \quad (\_, (\overline{ty_n})ref, \_) = methInfo(pc) \quad widRefConvertible(refs, ref)$$

The typing rule for `ireturn` is similar:

$$\frac{\boxed{\mathcal{P}(pc) = \texttt{ireturn}}}{pc \vdash \Pi[\mathcal{ST} + int]} \quad (\_, (\overline{ty_n})int, \_) = methInfo(pc)$$

A `return` instruction may be used to end an instance instantiation method `<init>`. In this case, remember that the object to be initialized by the `<init>` has a type $init(\_, \_)$ at the beginning of the `<init>` (see Section 8.7.2) and an invocation of another `<init>` within the `<init>` changes the type $init(\_, \_)$ into a normal class (see Section 8.7.3). Thus we make use of the typing rule for `return` to ensure that the `<init>` has really invoked another `<init>` by requiring that no $init(\_, \_)$ occurs in $\mathcal{L}$ or $\mathcal{S}$. Note that the condition has no effect if the method is not an `<init>`, since a type of the form $init(\_, \_)$ can only be introduced at the beginning of an `<init>`.

$$\frac{\boxed{\mathcal{P}(pc) = \texttt{return}}}{\begin{array}{l} pc \vdash_{\sqsubseteq} \Pi[\mathcal{LT}, \mathcal{ST}] \\ \text{No } init(\_, \_) \text{ occurs in } \mathcal{LT} \text{ or } \mathcal{ST} \end{array}}$$

A subtle point is that we do not require that $\mathcal{LT}$ and $\mathcal{ST}$ do not contain $unin(\_, \_)$. This means that a method can contain a `new` instruction, which creates an uninitialized object, but no invocation of an `<init>` on the uninitialized object. In practice, however, the uninitialized object is not very useful, since other typing rules ensure that the uninitialized object can neither be used within the current method nor passed to the invoking method.

8.8 Throwing and handling exceptions

An *exception* in the JVM is (represented as) an object, called an *exception object* or simply an *exception*, whose class must be *Throwable* or a subclass of it. An exception can be thrown by an `athrow` instruction or by the JVM itself.

As mentioned before, the code of a method is always associated with an exception table. An *exception table* is a list $\overline{(from_n, to_n, target_n, exncls_n)}$ of so-called `catch` *clauses*, where $from_i$, to_i and $target_i$ are addresses and $exncls_i$ are class names for $i = 1, \ldots, n$. The set of all paths from $from_i$ to to_i is called a `try` *clause*, where each path is inclusive on the $from_i$ side and exclusive on the to_i side. The $target_i$ is called the `catch` *clause target*. The class $exncls_i$ must be the class *Throwable* or a subclass of it. For an address pc in a method we use $Catch(pc)$ to denote the exception table associated with the method.

A `catch` clause $(from, to, target, exncls)$ is said to *catch* an exception if the exception is thrown during the execution of the `try` clause from $from$ to to and the class of the exception (object) is $exncls$ or a subclass of it.

Let $exnref$ be the reference to an exception (object). Then we formally define the function $catch(pc, exnref, \overline{(from_n, to_n, target_n, exncls_n)})$ as follows: if there is a `catch` clause among $\overline{(from_n, to_n, target_n, exncls_n)}$ that catches the exception $exnref$ thrown at pc, then the function yields the first `catch` clause that catches the exception; otherwise, the function yields $failure$.

For two addresses pc and pc' we define $InstrPC(pc, pc')$ as the function that yields the set of the addresses of all instructions in a path from pc to pc', where each path is inclusive on the pc side and exclusive on the pc' side.

8.8.1 The state transitions and the typing rules for exception throwing and handling

The JVM may throw an exception during the execution of an instruction. The whole process of exception throwing consists of a step of creating an exception (object) on the operand stack and a step of throwing the exception by changing the bit component in the runtime state into the state *throw*. Although this paper does not consider the details about the kinds of exceptions thrown by each instruction, we formalize how the exception throwing works as follows:

$$\frac{\begin{array}{l}\text{The JVM throws an exception at the address } pc\\ exnref \text{ is new in } \mathcal{H} \qquad \langle\cdots\rangle \text{ is a new object of } Throwable \text{ or a subclass ot it}\end{array}}{\begin{array}{l}\Xi[notThrow, pc, \mathcal{S}, \mathcal{H}]\\ \quad\Longrightarrow\ \Xi[throw, pc\mathcal{S} + exnref, \mathcal{H} \cup \{exnref \mapsto \langle\cdots\rangle\}]\end{array}}$$

In contrast, an `athrow` instruction does not create an exception (object): it only throws the given exception exception (object) as the top of the operand stack by changing the bit component into *throw*.

$$\frac{\mathcal{P}(pc) \equiv \texttt{athrow}}{\Xi[notThrow, pc] \implies \Xi[throw, pc]}$$

Since the above two state transitions are the only state transitions for exception throwing, it is easy to see that if the bit component has the state *throw*, then the operand stack contains an exception (object) to be thrown as the top entry.

The state *throw* indicates that an exception is being thrown and an exception handling is needed. We distinguish three cases. The first is when the exception is not caught by a `catch` clause of the method containing the current address. In this case, control is transferred to the last invoking method and the bit component in the runtime state remains to have the state *throw*.

$$\frac{catch(pc, exnref, Catch(pc)) = failure}{\begin{array}{l}\Xi[throw, pc, \mathcal{J} + (pc', \mathcal{L}', \mathcal{S}', \_), \mathcal{L}, \mathcal{S} + exnref, \mathcal{H}]\\ \implies \Xi[throw, pc', \mathcal{J}, \mathcal{L}', , \mathcal{S}' + exnref, \mathcal{H}]\end{array}}$$

The second case is when the exception is caught by a `catch` clause in the method containing the current address. In this case, control is transferred to the `catch` clause target and the bit component is changed into the state *notThrow*.

$$\frac{catch(pc, objref, Catch(pc)) = (\_, \_, target, \_)}{\Xi[throw, pc, \mathcal{J}, \mathcal{L}, \mathcal{S} + objref, \mathcal{H}] \implies \Xi[notThrow, target, \mathcal{J}, \mathcal{L}, [objref], \mathcal{H}]}$$

The third case is when the current method is the `main` method and there are no `catch` clauses catching the exception. In this case we require that the execution terminates for the sake of simplicity in this paper.

$$\frac{catch(pc, objref, Catch(pc)) = failure}{\Xi[throw, pc, [], \mathcal{L}, \mathcal{S} + objref, \mathcal{H}] \implies}$$

The typing rules specify only types for runtime states with the bit component state $notThrow$. In other words, the runtime states with the bit component state $throw$ are invisible from the typing point of view.

The typing rule for an `athrow` instruction checks only whether the exception on the operand stack has a correct type before the execution of the `athrow` instruction.

$$\frac{\boxed{\mathcal{P}(pc) \equiv \texttt{athrow}} \atop pc \vdash \Pi[\mathcal{ST} + refs] \qquad widRefConvertible(refs, Throwable)}{}$$

For each `catch` clause $(from, to, target, excncls)$, we need to impose the following typing rule for each instruction inside the `try`-clause from $from$ to to.

$$\frac{\boxed{\begin{array}{l} pc \in InstrPC(from, to) \text{ for a } \texttt{catch} \text{ clause } (from, to, target, excncls) \\ pc \vdash \Pi[\mathcal{ST}] \end{array}}}{target \vdash_{\sqsubseteq} \Pi([excncls])}$$

The JVM Specification (page 133) requires that a valid instruction sequence must not have an uninitialized object in a local variable in code protected by an exception handler. To describe the requirement, we could add to the above rule a restriction that neither $unin(\_,\_)$ nor $init(\_,\_)$ occur in the $\mathcal{LT}$ component of Π. However, test JVM programs show that the current Sun's implementation of Bytecode Verifier does not consider the restriction. Thus our specification does not consider the restriction, either.

The above rule could be refined by adding conditions analyzing the possible exception classes that may be thrown by a concrete instruction at pc and the possible `catch` clauses catching these exceptions. We do not do this for simplicity.

8.9 Implementing `finally`-clauses

The `jsr` and `ret` instructions are control transfer instructions typically used to implement `finally` clauses in `Java`. Following the JVM Specification, we call the address, to which a `jsr` instruction jumps, a `jsr` *target*, and the code starting from a `jsr` target a *subroutine*. Intuitively, a subroutine implements a `finally` clause, a `jsr` instruction calls a subroutine and a `ret` instruction returns from a subroutine to its caller. But a subroutine may also use other control transfer instructions to go to other addresses in the current method, so that for some programs it may be difficult to say where a subroutine is really ended.

8.9.1 State transitions for the `jsr` and `ret` instructions

The `jsr` instruction pushes its address pc onto the operand stack and transfers the control to $pc + \mathit{offset}(branch1, branch2)$.

$$\frac{\mathcal{P}(pc) = \texttt{jsr branch1 branch2}}{\Xi[pc, \mathcal{S}] \implies \Xi[pc + \mathit{offset}(branch1, branch2), \mathcal{S} + pc]}$$

Although it is not required by the specification, the first thing a subroutine does is typically to store the address of the `jsr` instruction in a local variable using an `astore` instruction.

A `ret` instruction uses the address stored in a local variable. If the local variable stores the address of a `jsr` instruction, then the `ret` instruction transfers the control to the address immediately after the `jsr` instruction.

$$\frac{\mathcal{P}(pc) = \texttt{ret index}}{\Xi[pc] \implies \Xi[\mathcal{L}(index) + 3]}$$

8.9.2 Typing the `jsr` and `ret` instructions

Typing the `jsr` and `ret` instruction is complex, since according to the JVM Specification they should have the following features:

- The same `ret` instruction can be reached from more than one `jsr` target. In other words, the same `ret` instruction can be shared by the codes of several subroutines. (In contrast, a return instruction can only be contained in the code of one method.)
- Not every path starting from a `jsr` target needs to reach a `ret` instruction. Furthermore, a `jsr` target need not reach a `ret` instruction. In other words, the code of a subroutine need not contain a `ret` instruction. A subroutine can be ended by a return instruction. The code of a subroutine can also be merged with the code of the method that calls it using control transfer instructions.
- A `jsr` target may reach another `jsr` target before it encounters a `ret` instruction. In other words, subroutines may be nested: a subroutine can call another subroutine. (This feature is essential to implement nested `finally` clauses.)
- A subroutine can call another subroutine and encounter a `ret` instruction that directly ends the calling subroutine. In this case, the `ret` instruction ends the called subroutine as well. (In contrast, a return instruction can only end the current method and return to the last invoking method.)
- An address can only be used at most once by a `ret` instruction. If a subroutine calls another subroutine, a `ret` instruction ends the calling subroutine and thus the called subroutine as well, then the return address of the called subroutine can no longer be used.

Technically, the mechanism is more complex. The reason is that a `finally` clause in a `Java` program often needs to be reached via totally different execution paths, e.g. a normal method execution and an exception throwing. Since each of these paths deals with its own values, it may happen that the value held by a particular local variable in a path is incompatible to the value held by the same local variable in another path. In this case, the usual typing rules would derive the *unusable* type for the local variable at the beginning of the `finally` clause. Thus the individual value assigned to the local variable in each path would be unusable within and, more important, after the `finally` clause. This would affect the original semantics of the `finally` clause.

To solve the problem, the JVM Specification suggests to turn off after a `finally` clause the usual typing process for all those local variables that are not used in the `finally` clause. Thus we need a mechanism to record those local variables that are used within a `finally` clause. The components $\mathcal{G}$, $\mathcal{B}$ and $\mathcal{U}$ in a state type have been reserved for this purpose. More precisely, we can define the meaning of "using a local variable within a `finally` clause" based on the derivation of these components in each typing rule. At the time being, we first define what these components are:

- A component $\mathcal{G}$ is a set of pairs of `jsr` targets. A pair $(target, target')$ in $\mathcal{G}$ should represent a directed edge, denoting a call of the subroutine starting with $target$ to the subroutine starting with $target'$. The set $\mathcal{G}$ should represent a directed acyclic graph, recording all `jsr` targets (i.e. the starting addresses of all nested subroutines) reaching the current address. We define $nodes(\mathcal{G}) := \{target \mid (target, \_) \in \mathcal{G} \text{ or } (\_, target) \in \mathcal{G}\}$.
- A component $\mathcal{B}$ is a subset of $nodes(\mathcal{G})$. It should consist of all `jsr` targets starting the closest subroutines containing the current address.

- A component $\mathcal{U}$ is a pair of mappings $(\{\overline{(from_g, to_g) \mapsto AccMod_g}\}, \{\overline{from'_h \mapsto AccMod'_h}\})$, where $\{\overline{(from_g, to_g)}\} \equiv \mathcal{G}$, $\{\overline{from'_h}\} \equiv \mathcal{B}$, $AccMod_i$ and $AccMod'_i$ are sets of indices of some local variables. Intuitively, $AccMod_i$ should consist of the indices of all local variables that may be used in an execution path from $from_i$ to to_i, and $AccMod'_i$ those from $from'_i$ to the current address.

For two state types $(\mathcal{LT}, \mathcal{ST}, \mathcal{G}, \mathcal{B}, \mathcal{U})$ and $(\mathcal{LT}', \mathcal{ST}', \mathcal{G}', \mathcal{B}', \mathcal{U}')$, we define the relation $\sqsubseteq$ on the newly defined components as follows:

- $\mathcal{G} \sqsubseteq \mathcal{G}'$ holds if and only if $\mathcal{G} \subseteq \mathcal{G}'$ holds.
- $\mathcal{B} \sqsubseteq \mathcal{B}'$ holds if and only if $\mathcal{B} \subseteq \mathcal{B}'$ holds.
- $\mathcal{U} \sqsubseteq \mathcal{U}'$ holds if and only if $\mathcal{U}(from, to) \subseteq \mathcal{U}'(from, to)$ holds for each $(from, to) \in \mathcal{G}$ and $\mathcal{U}(from) \subseteq \mathcal{U}'(from)$ for each $from \in \mathcal{B}$.

The definitions completely coincide with the intuitive meaning of $\sqsubseteq$ on state types (see Section 6).

Now let us explain the intuitive meaning of a +`jsr` address type $ad(pc)$. As mentioned in the state transition for `jsr`, a `jsr` instruction pushes the address pc' of a `jsr` instruction onto the operand stack and transfers the control to the subroutine. The subroutine may further store the address pc' from the operand stack in a local variable. In order to prevent the address pc' from illegal manipulations, we give the address pc' a special type $ad(pc)$, where the address pc in $ad(pc)$ is the `jsr` target. In general, a +`jsr` address type $ad(pc)$ contains addresses of all +`jsr` instructions whose target is pc.

There are two typing rules for a `jsr` instruction, formalizing the relationship between the type at the address of the `jsr` instruction and the type at the `jsr` target or the address immediately after the `jsr` instruction, respectively. The first rule is as follows:

$$\frac{\boxed{\begin{array}{l}\mathcal{P}(pc) = \texttt{jsr branch1 branch2} \\ pc \vdash \Pi[\mathcal{ST}, \mathcal{G}, \mathcal{B}, \mathcal{U}] \quad target = pc + offset(branch1, branch2)\end{array}} \\ target \notin nodes(\mathcal{G})}{\begin{array}{l}target \\ \vdash_{\sqsubseteq} \Pi[\mathcal{ST} + \{ad(target)\}, \mathcal{G} \cup \{(from', target) \mid from' \in \mathcal{B}\}, \{target\}, \mathcal{U} \cup \{target \mapsto \emptyset\}]\end{array}}$$

Note that the condition $target \notin nodes(\mathcal{G})$ ensures that no subroutines may be called recursively. The old $\mathcal{G}$ is extended with the edges $\{(from', target) \mid from' \in \mathcal{B}\}$ that represent calls of the old current subroutines to the new subroutine. The old $\mathcal{B}$ is replaced by one containing the `jsr` target $target$. The old $\mathcal{U}$ is extended at least with a mapping $target \mapsto \emptyset$, denoting the fact that no local variables have been used since the beginning $target$ of the subroutine.

Before turning to the second typing rule for `jsr`, let us first formalize the typing rule for `ret`, which is still quite simple.

$$\frac{\boxed{\mathcal{P}(pc) = \texttt{ret index} \quad pc \vdash \Pi[\mathcal{LT}]} \\ \mathcal{LT}(index) = ads \\ \text{if } \mathcal{P}(pc') = \texttt{ret index1},\ pc' \not\equiv pc \text{ and } pc' \vdash \Pi'[\mathcal{LT}'] \text{ then } \mathcal{LT}(index) \cap \mathcal{LT}'(index1) = \emptyset}{}$$

The rule enusres that the local variable $index$ really holds an address. In addition, we require that no two different `ret` instructions return from the same subroutine for the sake of simplicity in the specification. This is formalized as the condition $\mathcal{LT}(index) \cap \mathcal{LT}'(index1) = \emptyset$. Note that the condition is not a serious restriction, since whenever two `ret` instructions are needed, one can always write a `ret` and a `goto` instruction to the `ret`. The JVM Specification (page

124) requires the same condition without explaining why it is needed. We guess that it is just for the sake of simplicity in the implementation.

In order to formulate the second typing rule for `jsr`, we need more auxiliary functions.

First of all, the condition $\mathcal{LT}(index) \cap \mathcal{LT}'(index1) = \emptyset$ in the above rule guarantees the well-definedness of the auxiliary function $retOf(pc)$ defined as follows: for any address pc with $\mathcal{P}(pc) =$ `jsr branch1 branch2`, if there exists an address pc' with $\mathcal{P}(pc') =$ `ret index` such that $pc' \vdash \Pi'[\mathcal{LT}']$ and $ad(pc + offset(branch1, branch2)) \in \mathcal{LT}'(index)$, then $retOf(pc)$ yields the address pc'; otherwise it yields $failure$.

We define a $reachable$ relation on $nodes(\mathcal{G})$ as the smallest reflexive and transitive relation including that $reachable(target, target', \mathcal{G})$ for all $(target, target') \in \mathcal{G}$.

We define a function, which yields the set of all nodes in a path from a node $target \in nodes(\mathcal{G})$ to a node in a set $\{\overline{target_n}\} \subseteq nodes(\mathcal{G})$ as

$$\begin{array}{l} fromTo(target, \{\overline{target_n}\}, \mathcal{G}) \\ := \{target' \mid reachable(target, target', \mathcal{G}), reachable(target', target_i, \mathcal{G}) \text{ for some } 1 \leq i \leq n\}. \end{array}$$

The component $\mathcal{U}$ in a state type $(\mathcal{LT}, \mathcal{ST}, \mathcal{G}, \mathcal{B}, \mathcal{U})$ does not yield directly the set of the indices of all local variables that may be used in an execution path from a given `jsr` target to the current address. To do that we define the following function for $target \in nodes(\mathcal{G})$:

$$\begin{array}{l} used(target, \mathcal{G}, \mathcal{B}, \mathcal{U}) := \\ \begin{cases} \bigcup_{(target, target') \in \mathcal{G}} (\mathcal{U}(target, target') \cup used(target', \mathcal{G}, \mathcal{B}, \mathcal{U})) & target \notin \mathcal{B} \\ \mathcal{U}(target) & target \in \mathcal{B} \end{cases} \end{array}$$

As mentioned before, when a `ret` finishes a subroutine by transferring the control back to the calling `jsr`, we need to let the type of the address of the calling `jsr` and the types of the addresses of all calling `jsr` of the nested subroutines inside the current subroutine become $unusable$. In addition, since there may exist subroutines that merge into the code of the current subroutine but are invisible at the address of the calling `jsr` to the current subroutine, when returning from the current subroutine, we still need to eliminate the `jsr` address types for these subroutines. To do the above things, we define the function $filter(any, \{\overline{pc_n}\}, \{\overline{pc'_m}\})$ below. Intuitively, the set $\{\overline{pc_n}\}$ consists of all those `jsr` targets (i.e. the starting addresses for all those nested subroutines), from which the control is returned. The set $\{\overline{pc'_m}\}$ includes all those `jsr` targets (i.e. the starting addresses for all those subroutines) that can reach the address, to which the control is returned. Then the function changes the type or type set any with respect to $\{\overline{pc_n}\}$ and $\{\overline{pc'_m}\}$ as follows:

$$\begin{array}{rcll} filter(ads, \{\overline{pc_n}\}, \_) & := & unusable & \text{if } ads \cap \{\overline{ad(pc_n)}\} \neq \emptyset \\ filter(ads, \{\overline{pc_n}\}, \{\overline{pc'_m}\}) & := & ads \cap \{\overline{ad(pc'_m)}\} & \text{if } ads \cap \{\overline{ad(pc_n)}\} = \emptyset \\ filter(any, \_, \_\}) & := & any & \text{otherwise} \end{array}$$

Now we formulate the rule:

$$\frac{\boxed{\begin{array}{l} \mathcal{P}(pc) = \texttt{jsr branch1 branch2} \qquad \mathcal{P}(pc') = \texttt{ret index} \\ pc \vdash (\mathcal{LT}, \mathcal{ST}, \mathcal{G}, \mathcal{B}, \mathcal{U}) \qquad retOf(pc) = pc' \qquad pc' \vdash (\mathcal{LT}', \mathcal{ST}', \mathcal{G}', \mathcal{B}', \mathcal{U}'] \\ target = pc + offset(branch1, branch2) \qquad ad(target) \in \mathcal{LT}'(index) \end{array}}}{\begin{array}{l} pc + 3 \\ \vdash_{\sqsubseteq} (\mathcal{LT}[j \mapsto filter(\mathcal{LT}'(j), fromTo(target, \mathcal{B}', \mathcal{G}'), nodes(\mathcal{G})) \mid j \in used(target, \mathcal{G}', \mathcal{B}', \mathcal{U}')], \\ \quad \mathcal{ST}', \mathcal{B}, \mathcal{U}[from' \mapsto \mathcal{U}(from') \cup used(target, \mathcal{G}', \mathcal{B}', \mathcal{U}') \mid from' \in \mathcal{B}]) \end{array}}$$

The conditions in the box in the above rule ensures that the instruction `ret` at pc' really ends the subroutine that the `jsr` at pc calls. If it is not the case, then the rule does not apply. If

$\mathcal{LT}'(j)$ contains a `jsr` address type $ad(pc'')$ with a `jsr` target $pc'' \in fromTo(target, \mathcal{B}', \mathcal{G}')$, i.e. $\mathcal{LT}'(j)$ contains a `jsr` address type for a nested subroutine, then the formula

$$filter(\mathcal{LT}'(j), fromTo(target, \mathcal{B}', \mathcal{G}'), nodes(\mathcal{G}))$$

yields *unusable*; otherwise, the formula yields a set of `jsr` address types, where all `jsr` targets are in $\mathcal{G}$. The formula

$$\mathcal{LT}[j \mapsto \cdots \mid \ j \in used(target, \mathcal{G}', \mathcal{B}', \mathcal{U}')]$$

means that if a local variable has no possibilities to have been used since the address *target*, then its type at $pc + 3$ should cover its type at pc; otherwise, the type of a local variable at $pc+3$ should be the result of the previous formula. Note that the operand type stack at $pc+3$ should cover the $\mathcal{ST}'$ at the address pc' of the `ret` instruction. The formula

$$\mathcal{U}[from' \mapsto \mathcal{U}(from') \cup used(target, \mathcal{G}', \mathcal{B}', \mathcal{U}') \mid from' \in \mathcal{B}]$$

means that the set of used local variables at $pc + 3$ includes all used local variables at pc and all used local variables in the subroutine.

8.9.3 The $\mathcal{G}, \mathcal{B}$ and $\mathcal{U}$ components in other typing rules

This section changes/refines the typing rules in Sections 8.2-8.8 so that they take the components $\mathcal{G}, \mathcal{B}$ and $\mathcal{U}$ into consideration. Here we will discuss only the typing rules that need to be reconsidered. Those typing rules that are correct in their original forms will not be mentioned.

The first group of typing rules that need to be reconsidered are those load and store instructions (in Section 8.2) that load from or store into local variables, i.e. the `aload`, `iload`, `astore` and `istore` instructions. The changed rules for `aload` and `iload` can be formulated uniformly as follows:

$$\frac{\begin{array}{l}\boxed{\mathcal{P}(pc) \equiv \cdots \texttt{index} \qquad pc \vdash \Pi[\cdots, \mathcal{G}, \mathcal{B}, \mathcal{U}]} \\ \cdots\end{array}}{pc + 2 \vdash_{\sqsubseteq} \Pi[\cdots, \mathcal{G}, \mathcal{B}, \mathcal{U}[from' \mapsto \mathcal{U}(from') \cup \{index\} \mid from' \in \mathcal{B}]]}$$

where each $\cdots$ stands for the formulation at the corresponding place in the individual original rule. The intuitive meaning is that these rules use the local variable *index* and thus the index *index* should be added to the component $\mathcal{U}$.

The changed rules for `astore` and `istore` can be formulated uniformly in a very similar way:

$$\frac{\begin{array}{l}\boxed{\mathcal{P}(pc) \equiv \cdots \texttt{index}} \\ pc \vdash \Pi[\cdots, \mathcal{G}, \mathcal{B}, \mathcal{U}] \\ \cdots\end{array}}{pc + 2 \vdash_{\sqsubseteq} \Pi[\cdots, \mathcal{G}, \mathcal{B}, \mathcal{U}[from' \mapsto \mathcal{U}(from') \cup \{index\} \mid from' \in \mathcal{B}]]}$$

The second group of typing rules that need to be reconsidered are those for the beginning of a method body (in Section 8.7.2). All the original rules contain the initial components $\mathcal{G}_{start}, \mathcal{B}_{start}$ and $\mathcal{U}_{start}$. We need only to add the definitions $\mathcal{G}_{start} := \mathcal{B}_{start} := \mathcal{U}_{start} := \emptyset$, indicating that no subroutines have been called since the beginning of a method.

The third group of typing rules that need to be reconsidered are the first two typing rules for `invokespecial` (in Section 8.7.3), which deal with the invocation of an instance instatiation method `<init>`. The intuition is that a local variable may hold an uninitialized object and an invocation of `<init>` may change an uninitialized object in a local variable into an initialized one. In this case, the corresponding index of the local variable should be added to the component $\mathcal{U}$. The changed rules can be formulated uniformly as follows:

$$\frac{\begin{array}{l}\boxed{\mathcal{P}(pc) \equiv \texttt{invokespecial index1 index2}} \\ pc \vdash \Pi[\cdots, \mathcal{G}, \mathcal{B}, \mathcal{U}] \\ \cdots\end{array}}{pc + 3 \vdash_{\sqsubseteq} \Pi[\cdots, \mathcal{G}, \mathcal{B}, \mathcal{U}[\mathit{from}' \mapsto \mathcal{U}(\mathit{from}') \cup \mathcal{I} \mid \mathit{from}' \in \mathcal{B}]]}$$

where $\mathcal{I} := \{i \mid \mathit{unin}(pc', cls) = \mathcal{LT}(i)\}$ for the first typing rule, $\mathcal{I} := \{i \mid \mathit{init}(pc', cls') = \mathcal{LT}(i)\}$ for the second typing rule, and each $\cdots$ stands for the formulation at the corresponding place in the individual original rule.

9 Static well-typedness

In order to formulate the static well-typedness of the program $\mathcal{P}$, we define a *state type family* as a mapping $\{pc \mapsto \Pi_{pc} \mid pc \in PC(\mathcal{P})\}$. We use Φ to range over state type families.

We extend the relation $\sqsubseteq$ to the state type families as follows: Let Φ_1 and Φ_2 be two state type families. Then $\Phi_1 \sqsubseteq \Phi_2$ holds if and only if $\Phi_1(pc) \sqsubseteq \Phi_2(pc)$ hold for all $pc \in PC(\mathcal{P})$.

We define several satisfaction relations as follows:

- Φ is said to *satisfy* $pc \vdash \Pi$ if $\Pi \equiv \Phi(pc)$.
- Φ is said to *satisfy* $pc \vdash_{\sqsubseteq} \Pi$ if $\Pi \sqsubseteq \Phi(pc)$.
- Φ is said to *satisfy* a typing rule

$$\frac{\begin{array}{c}\boxed{\text{conditions1}} \\ \mathit{conditions2}\end{array}}{\mathit{conditions3}}$$

  if whenever Φ satisfies each condition of $\boxed{\text{conditions1}}$, Φ satisfies each condition in *conditions*2, and whenever Φ satisfies each condition in $\boxed{\text{conditions1}}$ and *conditions*2, Φ satisfies each condition in *conditions*3.

Now if a state type family Φ satisfies all typing rules in Section 8, then we say that the program $\mathcal{P}$ *has* the state type family Φ.

Finally, a program is said to be *statically well-typed* if it has a state type family.

There are still some questions that could be further studied here. For example, one may ask whether a statically well-typed program has a smallest state type family with respect to $\sqsubseteq$. Another question would be to ask how the specification is related to the theory of abstract interpretation [3, 4, 12]. We omit the discussions here.

10 Compile-time vs. runtime type-correctness

An advantage of a formal specification over an informal one is that one can formally state and prove properties of the formal specification. The JVM Specification requires in an informal way that nearly all type checking is done at compile time, and thus data need not be tagged with types. In our specification[2] we can formally prove that if a program is statically well-typed, then no type checking needs to be done at run time. Strictly speaking, the type inference system in our specification is still not a type checking algorithm. But it resembles the dataflow analysis

[2]Note that our specification considers only a subset of the whole JVM.

and abstract interpretation, and thus there is no problem to write one with respect to it. See e.g. the work [1, 3, 4, 12] for the details.

To perform the proof described above, we consider the following *(type) tags*:

$$tag \quad = \quad cls \mid Null \mid prim \mid returnAddress$$

These tags are in fact the JVM types given in the JVM Specification plus the type $Null$.

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

$$\begin{array}{rl}
correct(tag, any) \text{ holds} & \text{if } tag \text{ is } Null, cls \text{ or } prim \text{ and } tag \sqsubseteq any \\
correct(tag, refs) \text{ holds} & \text{if } tag \text{ is } Null \text{ or } cls \text{ and } widRefConvertible(tag, refs) \\
correct(cls, unin(_, cls)) \text{ holds} & \\
correct(cls, init(_, cls')) \text{ holds} & \text{if } widRefConvertible(cls, cls') \\
correct(returnAddress, ad(_)) \text{ holds} &
\end{array}$$

We explicitly associate each value with a unique tag using a mapping $tagOf$ of values to tags. For our specification values in a program must be created by some state transitions below. We define the mapping $tagOf$ with respect to these state transitions. For simplicity, we use the notations in the corresponding state transition rule in each case in the definition.

- For the state transition for `aconst_null`, define $tagOf(null) := Null$.
- For the state transition for `bipush byte`, define $tagOf(int\_extension(byte)) := int$.
- For the state transition for `iadd`, define $tagOf(val1 + val2) := int$.
- For the state transition for `new index1 index2`, define $tagOf(objref) := cls$. Furthermore, if $fldName$ is a field name of cls and ty is the type of $fldName$, then the object $\langle \cdots \rangle$ must be of the form $\langle \cdots, fldName = val, \cdots \rangle$, where val is a default value. We define $tagOf(val) := ty$ if ty is $prim$, $tagOf(val) := Null$ if ty is ref.
- For the state transition "the JVM throws an exception", define $tagOf(exnref) := exncls$, where $exncls$ is $Throwable$ or a subclass of it. Furthermore, if $fldName$ is a field name of $exncls$ and ty is the type of $fldName$, then the object $\langle \cdots \rangle$ must be of the form $\langle \cdots, fldName = val, \cdots \rangle$, where we define $tagOf(val) := ty$ if ty is $prim$, $tagOf(val) := Null$ if $val = null$ and ty is ref, and $tagOf(val)$ as ty or a subclass of it if none of both conditions above holds.
- We assume that all objects in the initial heap $\mathcal{H}_{start}$ at the starting address of the method `main` are also created as described in the above.
- For the state transition for `jsr branch1 branch2`, define $tagOf(pc) := returnAddress$.

For notational simplicity, we say

1. $correct(val, any)$ holds if and only if $correct(tagOf(val), any)$ does.
2. $correct(\mathcal{L}, \mathcal{LT})$ holds if and only if $\mathcal{L} = \{i \mapsto val_i \mid 0 \le i < n\}$, $\mathcal{LT} = \{i \mapsto any_i \mid 0 \le i < n\}$ and $correct(tagOf(val_i), any_i)$ hold for $i = 0, \ldots, n$.
3. $correct(\mathcal{S}, \mathcal{ST})$ holds if and only if $\mathcal{S} = [val_0, \cdots, val_n]$, $\mathcal{ST} = [usable_0, \cdots, usable_n]$ and $correct(tagOf(val_i), usable_i)$ hold for $i = 0, \ldots, n$.

We say that a heap $\mathcal{H}$ is *type correct*, or say that $correct(\mathcal{H})$ holds, if the following condition is true: for each $objref \mapsto \langle \cdots, fldName = val, \cdots \rangle \in \mathcal{H}$, if $tagOf(objref) = cls$, then $fldName$ is a field in the class cls such that if ty is the type declared for $fldName$ then $correct(val, ty)$ holds.

Since this paper assumes that a JVM program starts execution with a method `main`, which has no arguments or return values, we formally denote an execution of a JVM program as a sequence of state transitions in the following form

$$\Longrightarrow \Xi_1 \Longrightarrow \Xi_2 \Longrightarrow \cdots$$

where Ξ_1 is the state created by the transition rule for `main`.

As the last preparation, we have the following lemma:

Lemma 10.1 *Let* $(\mathcal{LT}, \mathcal{ST}, \mathcal{G}, \mathcal{B}, \mathcal{U})$ *be a state type at* $pc \in PC(\mathcal{P})$ *with* $target \in nodes(\mathcal{G})$. *Assume that an execution of a JVM program is of the form*

$$\Longrightarrow \Xi_1 \Longrightarrow \cdots \Longrightarrow \Xi_g \Longrightarrow \cdots \Longrightarrow \Xi_h$$

where Ξ_g *is of the form* $(notThrow, target, \_, \mathcal{L}, \_, \_)$ *and* Ξ_h *of the form* $(notThrow, pc, \_, \mathcal{L}', \_, \_)$. *Then* $\mathcal{L}(i) \neq \mathcal{L}'(i)$ *implies that* $i \in used(target, \mathcal{G}, \mathcal{B}, \mathcal{U})$.

Proof Follows from the definition of the function *used* (Section 8.9.2) and the constructions of the components $\mathcal{G}, \mathcal{B}$ and $\mathcal{U}$ in each typing rule. $\square$

Now we formulate the main result:

Theorem 10.2 *(Soundness) Let the program* $\mathcal{P}$ *have a state type family* Φ *and an execution*

$$\Longrightarrow \Xi_1 \Longrightarrow \cdots \Longrightarrow \Xi_k$$

where $k \geq 1$, $\Xi_k = (notThrow, pc_k, \_, \mathcal{L}_k, \mathcal{S}_k, \mathcal{H}_k)$ *and* $\Phi(pc_k) = (\mathcal{LT}_{pc_k}, \mathcal{ST}_{pc_k}, \_, \_, \_)$. *Then* $correct(\mathcal{L}_k, \mathcal{LT}_{pc_k})$, $correct(\mathcal{S}_k, \mathcal{ST}_{pc_k})$ *and* $correct(\mathcal{H}_k)$ *hold.*

Proof We assume that $\Xi_i = (bc_i, pc_i, \mathcal{J}_i, \mathcal{L}_i, \mathcal{S}_i, \mathcal{H}_i)$ and $\Phi(pc_i) = (\mathcal{LT}_{pc_i}, \mathcal{ST}_{pc_i}, \_, \_, \_)$ for all $i = 1, \ldots, k$. We prove the assertion of the theorem by an induction on k.

Let $k = 1$. By the state transition for `main`, we have $bc_1 = notThrow$, $\mathcal{L}_1 = \mathcal{L}^{max\_locals}$, $\mathcal{S}_1 = []$ and $\mathcal{H}_1 = \mathcal{H}_{start}$. By the typing rule for `main`, $\{i \mapsto unusable \mid 0 \leq i < max\_locals\} \sqsubseteq \mathcal{LT}_{pc_1}$ and $\mathcal{ST}_{pc_1} = []$ hold. Note that $correct(\mathcal{H}_{start})$ holds. Hence the assertion of the theorem holds.

We assume that $h \geq 1$, and as the induction assumtpion that the assertion of the theorem holds for $1 \leq k \leq h$. Now let $k = h + 1$. We assume that the execution

$$\Longrightarrow \Xi_1 \Longrightarrow \cdots \Longrightarrow \Xi_k$$

contains

$$\Xi_g \Longrightarrow \cdots \Longrightarrow \Xi_{h+1}$$

such that $g \leq h$, $bc_g = bc_{h+1} = notThrow$ and $bc_i = throw$ for all $g < i \leq h$. Note that except for exception throwing and handling we need only to consider the case that $g = h$, i.e. the sequence is $\Xi_h \Longrightarrow \Xi_{h+1}$.

If $\Xi_h \Longrightarrow \Xi_{h+1}$ is for `aload`, `iload`, `astore`, `istore`, `aconst_null`, `bipush`, `getfield`, `putfield`, `iadd`, `new`, `dup`, `pop`, `swap`, `if_acmpeq`, `if_icmpeq` or `goto`, then the proof is straightforward: the proofs follow from the corresponding typing rules and the induction assumption, except that the proofs for `aconst_null`, `bipush`, `iadd` and `new` additionally use the definition of $tagOf$.

If $\Xi_h \implies \Xi_{h+1}$ is for `invokespecial`, `invokevirtual`, `invokeinterface` or `invokestatic`, then the proof can be made based on the typing rules for the starting addresses of method bodies (Section 8.7.2).

If $\Xi_h \implies \Xi_{h+1}$ is a state transition for `areturn`, `ireturn` or `return`, then we know that the execution is of the form

$$\implies \Xi_1 \implies \cdots \implies \Xi_p \implies \Xi_{p+1} \implies \cdots \implies \Xi_h \implies \Xi_{h+1}$$

where $p < h$ and $\Xi_p \implies \Xi_{p+1}$ is the last state transition for an invocation instruction. Assume that $\mathcal{S}_p$ is of the form $\mathcal{S}[+objref]\overline{+val_n}$, where $[+objref]$ means that the presence of $objref$ depends on whether the method is `static`. Then both $\mathcal{J}_{p+1}$ and $\mathcal{J}_h$ are the same and of the form $\mathcal{J}_p + (pc_p, \mathcal{L}_p, \mathcal{S}, \_)$. Furthermore, we have $\mathcal{L}_{h+1} = \mathcal{L}_p$, $\mathcal{S}_{h+1} = \mathcal{S}$ and $\mathcal{H}_{h+1} = \mathcal{H}_h$ by the state transition for `areturn`, `ireturn` or `return`. Let $\Phi(pc_p) = (\mathcal{LT}_{pc_p}, \mathcal{ST}[+tys]\overline{+tys_n}, \_, \_, \_)$, where the presence of $+tys$ depends on whether the method is `static`. By the typing rules for method invocations (Section 8.7.3), $\mathcal{LT}_{pc_p} \sqsubseteq \mathcal{LT}_{pc_{h+1}}$ and $\mathcal{ST} \sqsubseteq \mathcal{ST}_{pc_{h+1}}$ hold. By the induction assumption, $correct(\mathcal{L}_p, \mathcal{LT}_{pc_p})$, $correct(\mathcal{S}, \mathcal{ST})$ and $correct(\mathcal{H}_h)$ hold. Hence $correct(\mathcal{L}_{h+1}, \mathcal{LT}_{pc_{h+1}})$, $correct(\mathcal{S}_{h+1}, \mathcal{ST}_{pc_{h+1}})$ and $correct(\mathcal{H}_{h+1})$ hold.

If $\Xi_h \implies \Xi_{h+1}$ is for exception handling, then $\Xi_g \implies \cdots \implies \Xi_{h+1}$ is such that $bc_g = bc_{h+1} = notThrow$ and $bc_i = throw$ for all $g < i \leq h$, $\Xi_g \implies \Xi_{g+1}$ is a state transition for `athrow` or a JVM exception throwing and $\Xi_h \implies \Xi_{h+1}$ is the state transition with the condition $catch(pc, objref, Catch(pc)) = (\_, \_, target, \_)$. If $g + 1 = h$, then the assertion of the theorem directly follows from the induction assumption, the definition $tagOf$ on the exception object created by the JVM and the typing rule with the condition "$pc \in InstrPC(from, to)$ for $(from, to, target, exncls)$". If $g + 1 < h$, then $\Xi_{g+1} \implies {}^*\Xi_h$ consists of several steps of the state transition for the condition $catch(pc, objref, Catch(pc)) = failure$. In this case, the sequence of the state transitions is of the form

$$\implies \Xi_1 \implies \cdots \implies \Xi_p \implies \Xi_{p+1} \implies \cdots \implies \Xi_g \implies \Xi_{g+1} \implies \cdots \implies \Xi_h \implies \Xi_{h+1}$$

where $p < g$ and $\Xi_p \implies \Xi_{p+1}$ is the $(h - g - 1)$th last state transition for an invocation instruction. By a discussion similar to that for the state transition for `areturn`, `ireturn` or `return`, we can prove the assertion of the theorem using the induction assumption, the definition $tagOf$ on the exception object created by the JVM and the typing rule with the condition "$pc \in InstrPC(from, to)$ for $(from, to, target, exncls)$".

If $\Xi_h \implies \Xi_{h+1}$ is for `jsr`, then the proof follows directly from the first typing rule for `jsr`.

If $\Xi_h \implies \Xi_{h+1}$ is for `ret`, then the proof follows from the second typing rule for `jsr`, the induction assumption on the state transition for the corresponding `jsr` instruction and Lemma 10.1. □

11 Some other properties

In the previous section we have stated and proved a main property of our specification. In this section we state and prove a few more. Note that these properties here are just examples. We cannot consider all important properties due to lack of space.

Proposition 11.1 *Let $pc \in PC(\mathcal{P})$ and Φ_1 and Φ_2 be two arbitrary state type families such that $\Phi_1 \sqsubseteq \Phi_2$. Assume that $\Phi_i(pc)$ is of the form $(\mathcal{LT}_i, \mathcal{ST}_i, \_, \_, \_)$ for $i = 1, 2$.*

- *If $\mathcal{LT}_1 = prim$, raw or $unusable$, then $\mathcal{LT}_2 = \mathcal{LT}_1$ or $unusable$. If $\mathcal{ST}_1$ contains $prim$, raw or $unusable$ as an entry, then $\mathcal{ST}_2$ contains $\mathcal{LT}_1$ or $unusable$ at the same position.*
- *If $\mathcal{LT}_1 = ads$, then $\mathcal{LT}_2 = ads'$ such that $\mathcal{LT}_1 \subseteq \mathcal{LT}_2$, or $\mathcal{LT}_2 = unusable$.*

Proof Follows directly from the construction of the types, type sets and the definition of the relation $\sqsubseteq$. □

As mentioned before, the typing rules for an instruction specify precisely how the instruction behaves on an uninitialized object. The following proposition summarizes some of the results:

Proposition 11.2

1. *A reference to an object created by a* `new` *instruction can only be manipulated by the following instructions:* `aload`, `astore`, `dup`, `pop`, `swap`, `invokespecial`. *In the case of* `invokespecial`, *the method is* `<init>` *and the object is the one to be initialized and must be of the same class as the* `<init>`.

2. *Inside a method* `<init>` *that is not declared in the class Object, the object to be initialized can only be used by the following instructions:* `aload`, `dup`, `pop`, `swap`, *before an* `invokespecial` *for another* `<init>` *is called on the object, where the called* `<init>` *is declared in the same class as or in an immediate superclass of that of the object.*

3. *Inside a method* `<init>` *that is not declared in the class Object, there must be a call to an* `invokespecial` *for another* `<init>` *on the object.*

Proof The proofs for 1 and 2 follow directly from the typing rules for instructions. The proof for 3 follows from the constraints in the typing rules for the second typing rule for `invokespecial` and the typing rule for `return`. □

Proposition 11.3 *An address can be manipulated as a value only in the following ways:*

1. *It can be stored onto the operand stack by an* `jsr` *instruction.*

2. *It can be stored from the operand stack in a local variable by an* `astore` *instruction.*

3. *It can be manipulated in the operand stack by a* `dup`, `pop` *or* `swap` *instruction.*

4. *In a local variable, it can be used to compute the return address and made invalid by a* `ret` *instruction.*

Proof The proofs follow directly from the typing rules for instructions. □

Proposition 11.4 *Assume that a method invokes another method. Then the invoked method can never pass an uninitialized object back to the invoking method.*

Proof There are only two possibilities to transfer the control from an invoked method to its invoking method: executing a return instruction and throwing an exception. The state transitions for return instructions (Sections 8.7.4) and for exception handlings (Section 8.8) ensure that in both cases all data in the operand stack and local variables are erased, except the returning object or the exception object. The typing rules for the `areturn` instruction (Section 8.8), for state transition "the JVM throws an exception" and the typing rule for `athrow` (Section 8.8) ensure that both returning and exception object itself cannot be uninitialized. Furthermore, a returning or exception object cannot contain an uninitialized object for two reasons. First, the state transitions for an `new` instruction and for "the JVM throws an exception" ensures that a newly created object cannot contain an uninitialized object. Second, the typing rule for `putfield` instruction ensures that no uninitialized objects can be put into an object. Hence the assertion of the proposition holds. □

Proposition 11.5 *Assume that a method invokes another method that is not an* `<init>`*. Then the invoking method can never pass an uninitialized object to the invoked method.*

Proof The only possibility to pass a value from the invoking method to the invoked method is the invocation instruction. By the typing rules for method invocation instructions (Sections 8.7.3), we know that if the method is not an `<init>`, then the object, on which the method is invoked, and the parameters of the method cannot have type $unin(\_,\_)$ or $init(\_,\_)$. Furthermore, by a proof similar to that for the previous proposition, the object and these parameters cannot contain an uninitialized object. Thus the assertion of the proposition holds. □

12 Conclusion

We have shown a formal specification of a large subset of JVM instructions. Thanks to the way types and type sets are defined, the formal specification can be formulated precisely and simply. The formal specification clarifies several ambiguities and incompleteness and removes some (in our view) unnecessary restrictions in the description of the (informal) Java Virtual Machine Specification [8].

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 Vertual Machine specification. Technical report, University of Cambridge Computer Laboratory, 1997.

[3] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs bu construction of approximation of fixpoints. In *Proc. 4th ACM Symp. Principles of Programming Languages*, pages 238–258, 1977.

[4] P. Cousot and R. Cousot. Systematic design of data flow analysis frameworks. In *Proc. 6th ACM Symp. Principles of Programming Languages*, 1979.

[5] D. Dean. The security of static typing with dynamic linking. In *Proc. 4th ACM Conf. on Computer and Communications Security.* ACM, 1996.

[6] 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.

[7] J. Gosling, B. Joy, and G. Steele. *The Java™ Language Specification.* Addison-Wesley, 1996.

[8] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification.* Addison-Wesley, 1996.

[9] T. Nipkow and D. von Oheimb. $\text{Java}_{\ell ight}$ is type-safe — definitely. Submitted for publication, 1997.

[10] V. Saraswat. Java is not type-safe. Technical report, AT&T Research, 1997.

[11] D. Syme. Proving Java type soundness. Technical report, Computational Logic inc., 1997. Draft version.

[12] R. Wilhelm and D. Maurer. *Compiler Design.* Addison-Wesley Publishing Company, 1995.