Semantics of Java Byte Code

Peter Bertelsen, C917023

March 31, 1997

Contents

1 Introduction

This report describes the results of a student project entitled *Semantics of Java Byte Code*, arranged as a 'special course' at the Department of Information Technology, Technical University of Denmark, February through March 1997.

Associate professor Hans Bruun (Department of Information Technology, Technical University of Denmark), and associate professor Peter Sestoft (Department of Mathematics and Physics, Royal Veterinary and Agricultural University in Copenhagen) have functioned as tutors.

The purpose of the project has been that of studying the semantics of Java byte code, and hence of the Java Virtual Machine, and that of developing a formal specification of the semantics. The formal specification is based on the official, informal specification in *The Java Virtual Machine Specification*[2], published by Addison-Wesley on behalf of Sun Microsystems, Inc.

The reader is assumed to be familiar with the Java programming language, and with object-oriented terminology in general. The reader is also encouraged to study Chapter 2 through 4 of *The Java Virtual Machine Specification*[2], since these provide a good introduction to the Java Virtual Machine: Chapter 2 gives an introduction to related parts of the semantics of Java; Chapter 3 describes the structure of the Java Virtual Machine; and Chapter 4 provides a reasonably precise specification of the format of Java class files.

The Java Virtual Machine is a multi-threaded stack machine, supporting synchronization amongst multiple threads of execution. This report, however, only describes the semantics of a single thread of execution. In other words, the Java Virtual Machine is viewed as a single-threaded state machine, and synchronization primitives are not specified.

The structure of this report is as follows:

- Section 2 describes the formal specification of the semantics of the Java Virtual Machine
- Section 3 describes a few ambiguities in *The Java Virtual Machine Specification*[2]
- Section 4 concludes, and
- Appendix A lists minor errors in *The Java Virtual Machine Specification*[2].

2 Java Virtual Machine Semantics

In the following sections we present a formal specification of the semantics of the Java Virtual Machine. The formal specification is based on *The Java Virtual Machine Specification*[2], and on a few necessary assumptions (cf. Section 3).

2.1 An Example

In the following sections the semantics of the Java Virtual Machine is specified in terms of a set of transition rules, defining how the execution of an instruction changes the state of the machine, and under which circumstances this will happen. A transition rule

$$\frac{\begin{array}{l} instr(ts) = \texttt{dup} \\ s = (v : W) :: sr \\ size(s) + size(v) \leq max_s(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', v :: v :: sr, l, m) :: fr, h, e)}$$

defines the semantics of a single instruction, in this case the stack manipulation instruction `dup` which duplicates the topmost value on the operand stack (cf. Section 2.5.4, rule 5).

The propositions above the horizontal line are the premises of the rule; the proposition below the line is the conclusion of the rule. If all of the premises hold, the state of the Java Virtual Machine will change from the current state, ts, to the state specified after the $\Rightarrow$ symbol.

As a convention the symbols s, l, m, fr, h, and e are considered to be the names of the components of the thread state $ts = ((pc, s, l, m) :: fr, h, e)$; see Section 2.5.

Another convention is that the premises of the transition rules are to be read top-down, and left-right. The proposition $s = (v : W) :: sr$ in the above example is thus an assertion on, and binding of names to, the components of the operand stack: s holds at least one (topmost) value called v; the rest of the stack is called sr.

Similarly, the proposition $succ(ts) = pc'$ asserts that the semantic function *succ* is defined for ts; the value returned by that function is called pc'.

Semantic utility functions, such as *size*, are defined in Section 2.4.2. Semantic functions for extracting components of a thread state, e.g. *instr*, max_s, and *succ*, are defined in Section 2.4.3.

A special notation (borrowed from Standard ML) is used for matching values with certain types. In the above example the term $v : W$ specifies that the value v must be of type W. This notation is also used for tagging values with a certain type, e.g. $0 : Int$ denotes the value zero of type *Int*.

The semantic types used in this specification are defined in Section 2.3.

The notation used in the specification is further explained in the following section.

2.2 Notation

- The term $\mathcal{P}(X)$ denotes the power set of X: $\mathcal{P}(X) \equiv \{S \mid S \subseteq X\}$
- In some connections we use 'optional' types, denoted as: $[X] \equiv X \cup \{\texttt{none}\}$
- The operator $\cup$ is used for disjoint sum; i.e., $X \cup Y$ denotes the disjoint sum of X and Y.

- A finite map $f \in Y \to Z$ is considered a set of pairs $(y, z) \in Y \times Z$, where

  $$(y, z_1), (y, z_2) \in f \Rightarrow z_1 = z_2$$

  The elements of a finite map will generally be written $y \mapsto z$.

- The operators *dom* and *rng* denote the usual domain and range operations:

  $$\begin{array}{rcl} dom(X) & \equiv & \{y \mid (y \mapsto z) \in X\} \\ rng(X) & \equiv & \{z \mid (y \mapsto z) \in X\} \end{array} \qquad \text{where } X \in (Y \to Z); y \in Y; z \in Z$$

- For a finite map X the term X_y denotes the restriction of X with respect to y:

  $$X_y \equiv \{(y' \mapsto z) \in X \mid y' \neq y\}$$

  where $X \in (Y \to Z); y, y' \in Y; z \in Z$

- The operator + is used for merging two finite maps:

  $$X + X' \equiv \{(y \mapsto z) \in X \mid y \notin dom(X')\} \cup X'$$

  where $X, X' \in (Y \to Z); y \in Y; z \in Z$

- The operator :: is used for adding one element to a list or sequence:

  $$x :: <x_1, x_2, x_3, \ldots, x_n> \equiv <x, x_1, x_2, x_3, \ldots, x_n>$$

- Terms of the form $\{e(x, y) \mid f(x) = y\}$ are used as abbreviations for

  $$\{e(x, y) \mid x \in dom(f) \wedge f(x) = y\}$$

- An undefined expression e is considered unequal to any expression e'; in other words, $e = e'$ is false if e or e' is undefined.

2.3 Semantic Types

The following abstract types are used in this specification: Id_c, Id_f, Id_m, *Int*, *Long*, *Float*, *Double*, *String*, *Ref*, *Index*, *PC*, and *Offset*.

Id_c is the type of qualified class or interface names, whereas Id_f and Id_m are the types of unqualified field and method names, respectively.

The numeric types *Int*, *Long*, *Float*, and *Double* correspond to the simple Java types `int`, `long`, `float`, and `double`, respectively.

String is the type of (constant) character sequences; in a concrete class file, values of type *String* are stored in UTF-8 format (cf. [2, p. 100]). Note that type *String* is not the same as the Java class type `java.lang.String`.

Ref is the type of references to objects (class instances and array objects).

Index is the type of indices into the constant pool of a class file.

PC is the type of program counters (addresses of instructions), and *Offset* is the type of 'distances' between instruction addresses.

Other basic types include $\mathcal{B}$ (boolean values), $\mathcal{N}$ (natural numbers), and $\mathcal{N}_0$ (non-negative integers).

2.3.1 Class File Contents

The following type definitions represent an abstract view of the concrete structure of a Java class file:

$$\begin{aligned} C &= \mathcal{P}(Acc_c) \times [Id_c] \times \mathcal{P}(Id_c) \times FD \times CV \times MD \times MI \times CP \\ Acc_c &= \{\texttt{public}, \texttt{final}, \texttt{super}, \texttt{interface}, \texttt{abstract}\} \end{aligned}$$

The components of a class or interface declaration C comprise a set of access modifiers, the name of the direct superclass (optional), the names of direct superinterfaces, field declarations, constant values, method declarations, method implementations, and a constant pool.

The access modifier `interface` specifies that the declaration is for an interface; if the `interface` modifier is not present the declaration is for a class. The `super` modifier only affects the semantics of the `invokespecial` instruction (cf. Section 2.5.13, rule 52).

The direct superclass must be specified in a class declaration, unless the class being declared is `java.lang.Object` which has no superclass; the direct superinterfaces of a class are the interfaces that the class is declared to implement.

The direct superclass in an interface declaration must always be `java.lang.Object`; the direct superinterfaces of an interface are the interfaces that the interface is declared to extend (inherit from).

The set of field declarations in a class or interface declaration maps each declared field name to the access modifiers and type descriptors for that field:

$$\begin{aligned} FD &= Id_f \rightarrow (\mathcal{P}(Acc_f) \times Desc) \\ Acc_f &= \{\texttt{public}, \texttt{private}, \texttt{protected}, \texttt{static}, \texttt{final}, \texttt{volatile}, \texttt{transient}\} \\ Desc &= Simple \cup Array \cup Id_c \\ Simple &= \{\texttt{boolean}, \texttt{char}, \texttt{byte}, \texttt{short}, \texttt{int}, \texttt{long}, \texttt{float}, \texttt{double}\} \\ Array &= \mathcal{N} \times (Simple \cup Id_c) \end{aligned}$$

At most one of the `public`, `private`, and `protected` modifiers can be used for any field.

The type descriptor of a field can either be the name of a simple type, an array type descriptor, or the name of a class or interface type. An array type descriptor consists of the number of dimensions, and a base type descriptor. The base type descriptor of an array type can either be the name of a simple type, or the name of a class or interface type.

Class fields have the modifier `static`, whereas instance fields do not have the `static` modifier. Fields declared in an interface must be `static`.

The set of constant values in a class or interface declaration maps the name of each field with a constant value to the values for that field:

$$\begin{aligned} CV &= Id_f \rightarrow Const_v \\ Const_v &= Int \cup Long \cup Float \cup Double \cup String \end{aligned}$$

The constant values are used as initial values of the corresponding fields in connection with initialization of the class or interface (cf. Section 2.5.16). Constant values may only be specified for `static` fields, and must be assignment compatible with the type of the corresponding fields.

The set of method declarations in a class or interface declaration maps the method signature of each declared method to the access modifiers, return type descriptor (optional), and potential exceptions for that method:

$$\begin{aligned} MD &= Sig \rightarrow (\mathcal{P}(Acc_m) \times [Desc] \times \mathcal{P}(Id_c)) \\ Sig &= Id_m \times Desc^* \\ Acc_m &= \{\texttt{public}, \texttt{private}, \texttt{protected}, \texttt{static}, \texttt{final}, \texttt{synchronized}, \texttt{native}, \\ &\quad \texttt{abstract}\} \end{aligned}$$

A method signature consists of a method name, and a (possibly empty) list of parameter type descriptors. Constructors have the special method name `<init>`, whereas static initializers for classes and interfaces have the special method name `<clinit>`.

Note that the angular brackets are part of the method names for constructors and static initializers, and thus should not be mistaken for singleton lists.

At most one of the `public`, `private`, and `protected` modifiers can be used for any method. The `abstract` modifier can only be used for methods declared in an interface or `abstract` class. Methods declared in an interface must have the `abstract` modifier.

Constructors, static initializers, and methods declared to return no value (`void` methods in Java) do not have a return type descriptor.

The set of potential exceptions in a method declaration is the names of exception classes that the method is declared to throw; this does not imply that the method will in fact throw any exception, nor that it will throw only objects of those exception classes that it is declared to throw[1].

The set of method implementations in a class declaration maps the signature of each implemented method to the stack limit, local variable limit, code array, and exception handlers for that method:

$$\begin{aligned} MI &= Sig \to Impl \\ Impl &= \mathcal{N}_0 \times \mathcal{N}_0 \times Code \times H \\ Code &= PC \to Instr \\ Instr &= \{\texttt{aaload}, \texttt{aastore}, \texttt{aconst\_null}, \ldots\} \\ H &= PC \to \mathcal{P}(Id_c) \to PC \end{aligned}$$

An interface cannot implement any methods; hence its set of method implementations must be empty. A method declared to be `abstract` in a class declaration cannot also be implemented by the same class. Any method declared without the `abstract` modifier must be implemented by the same class in which it is declared.

The code array in a method implementation maps each program counter value (instruction address) to the instruction at that address. The complete instruction set of the Java Virtual Machine is listed in Section 2.5.1.

The set of exception handlers in a method maps a program counter value to another finite map; the latter maps names of exception classes to the initial program counter (address of the first instruction) for the corresponding handler. The parameter of the second 'level' of exception handlers is the set of classes that the exception is an instance of. See also the specification the instruction `athrow` in Section 2.5.17, and the explanation of the utility function *throw* on page 16.

The constant pool in an interface or class declaration maps a constant pool index to a constant value or symbolic reference:

$$\begin{aligned} CP &= Index \to (Const_v \cup Const_r) \\ Const_r &= Array \cup Id_c \cup Const_f \cup Const_m \cup Const_{im} \\ Const_f &= Id_c \times Id_f \times Desc \\ Const_m &= Id_c \times Sig \times [Desc] \\ Const_{im} &= Id_c \times Sig \times [Desc] \end{aligned}$$

[1] The Java Virtual Machine does not use the set of potential exceptions, but every method implementation must include the set of exceptions that the method is declared to throw

The constant values in a constant pool can be loaded onto the run-time operand stack by the instructions `ldc`, `ldc_w`, and `ldc2_w` (cf. Section 2.5.3).

Symbolic references are either to an array type, to a class or interface type, to a field, to a non-interface method, or to an interface method.

A reference to a field includes the name of the class or interface declaring the field, the name of the field, and the type descriptor for the field.

A reference to a method (non-interface as well as interface) includes the name of the class/interface declaring the method, the signature of the method, and the (optional) return type descriptor for the method.

All accesses to members of a class or interface must be via symbolic references, even for members of the same class as the referring method.

2.3.2 Thread State

In this specification the Java Virtual Machine is viewed as a single-threaded, infinite state machine. The run-time state of the machine is defined in terms of the components of the type *TS*:

$$TS = Frame^* \times Heap \times Env$$

The components of a thread state comprise a frame stack, a heap, and an environment. Each frame in the frame stack corresponds to the invocation of a method. The topmost frame in the frame stack (at the head of the list of frames) is considered the 'current' frame, corresponding to the local state of the method currently being executed.

A frame consists of a program counter, an operand stack, a set of local variables, and a current method identifier:

$$\begin{aligned} Frame &= PC \times Oper^* \times Locals \times (Id_c \times Sig) \\ Oper &= W \cup DW \\ W &= W_v \cup PC \\ W_v &= Int \cup Float \cup Ref_0 \\ Ref_0 &= Ref \cup \{\texttt{null}\} \\ DW &= Long \cup Double \\ Locals &= \mathcal{N}_0 \rightarrow Oper \end{aligned}$$

The program counter in a frame specifies the address of the current instruction for the method of the frame.

The operands in the operand stack are either one-word values or two-word values. A one-word operand is either a value of the simple types *Int* or *Float*, a reference to an object in the heap of the thread state, the special value `null`, or a program counter. A two-word operand is a value of the simple types *Long* or *Double*.

The local variables in a frame maps a non-negative integer index to the one-word or two-word value of the local variable at that index. Two-word values occupy two local variable entries each (cf. Section 2.5.5).

The method identifier in a frame consists of the name of the class declaring the method, and the signature of the method.

The heap in a thread state maps a reference to an object:

$$\begin{aligned} \mathit{Heap} &= \mathit{Ref} \rightarrow \mathit{Obj} \\ \mathit{Obj} &= \mathit{Obj}_u \cup \mathit{Obj}_c \cup \mathit{Obj}_a \\ \mathit{Obj}_u &= \mathit{Id}_c \times \mathit{IV} \\ \mathit{Obj}_c &= \mathit{Id}_c \times \mathit{IV} \\ \mathit{IV} &= (\mathit{Id}_c \times \mathit{Id}_f) \rightarrow V \\ V &= W_v \cup \mathit{DW} \\ \mathit{Obj}_a &= \mathit{Array} \times \mathit{Int} \times \mathit{AV} \\ \mathit{AV} &= \mathit{Int} \rightarrow V \end{aligned}$$

Note that the special value `null` is not in the domain of any heap.

An object is either an uninitialized instance of a class type, an initialized instance of a class type, or an (initialized) instance of an array type. A heap never contains any uninitialized objects of array type.

An object of class type consists of the name of the class, and a set of instance field values. The set of instance field values maps a class name and field name to the value of that instance field. The set of instance field values holds the values of instance fields declared in all classes that the object is an instance of (i.e., the class of the object, and all superclasses of that class).

An object of array type consists of the descriptor for the array type, a non-negative number specifying the length of the array, and a set of array components. The set of array components maps a non-negative integer index to the value of the array component at that index. The components of a multi-dimensional array are references to sub-arrays of the component array type (or the special value `null`, in case only the first dimension of the array has been initialized).

Note that values of type *PC* cannot be stored in any instance field or array component.

In a concrete implementation of the Java Virtual Machine, unused heap space is assumed to be reclaimed by a garbage collector. The issue of garbage collection is not addressed any further in this specification of the Java Virtual Machine semantics; instead, values of type *Heap* are assumed to have unlimited size.

The environment in a thread state maps a class or interface name to the declaration and static field values for that class or interface:

$$\begin{aligned} \mathit{Env} &= \mathit{Id}_c \rightarrow (C \times \mathit{SV}) \\ \mathit{SV} &= \mathit{Id}_f \rightarrow V \end{aligned}$$

The set of static field values for a class or interface maps a field name to the value of that field.

2.4 Semantic Functions

In the following sections we define a few abstract semantic functions, utility functions, thread state query functions, and predicates to be used in the specification of the semantics of the Java Virtual Machine instructions.

The rule numbers referred to in the following sections are the numbers of transition rules for instructions described in Section 2.5.

2.4.1 Abstract Semantic Functions

In this section we define a set of abstract semantic functions. The precise semantics of these functions is not considered any further in this specification of the Java Virtual Machine semantics.

The following total semantic functions are used for truncating values of type *Int* to 'smaller' Java types:

$$i2z, i2b, i2c, i2s : Int \to Int$$

The above functions truncate their *Int* operands to values suitable for a Java variable of type `boolean`, `byte`, `char`, and `short`, respectively. Note that these 'small integer' types are not used in the operand stack and local variables of the Java Virtual Machine; all values of the smaller types are represented as *Int* values, except when they are stored in arrays (cf. rule 20 ff).

These total semantic functions are used for converting between and truncating values of (abstract) numeric types:

$$\begin{aligned}
i2l &: Int \to Long \\
i2f &: Int \to Float \\
i2d &: Int \to Double \\
l2i &: Long \to Int \\
l2f &: Long \to Float \\
l2d &: Long \to Double \\
f2i &: Float \to Int \\
f2l &: Float \to Long \\
f2d &: Float \to Double \\
d2i &: Double \to Int \\
d2l &: Double \to Long \\
d2f &: Double \to Float
\end{aligned}$$

The following total arithmetic functions operate on values of (abstract) numeric types:

$$\begin{aligned}
&iadd, imul, isub, iand, ior, ixor, ishl, ishr, iushr : (Int \times Int) \to Int \\
&idiv, irem : (Int \times Int') \to Int \\
&ineg : Int \to Int \\
&ladd, lmul, lsub, land, lor, lxor : (Long \times Long) \to Long \\
&ldiv, lrem : (Long \times Long') \to Long \\
&lneg : Long \to Long \\
&lshl, lshr, lushr : (Long \times Int) \to Long \\
&fadd, fdiv, fmul, frem, fsub : (Float \times Float) \to Float \\
&fneg : Float \to Float \\
&dadd, ddiv, dmul, drem, dsub : (Double \times Double) \to Double \\
&dneg : Double \to Double
\end{aligned}$$

where $Int' = Int \backslash \{0 : Int\}$ and $Long' = Long \backslash \{0 : Long\}$

Note that the binary functions *fdiv* and *frem* (division and remainder) are defined for all values of type *Float*; similarly, *ddiv* and *drem* are defined for all values of type *Double*. If the second operand of either of these functions is zero, the result will be `NaN`.

These total semantic functions are used for comparing values of (abstract) numeric types:

$$\begin{aligned}
lcmp &: (Long \times Long) \to \{-1, 0, 1\} \\
fcmp &: (Float' \times Float') \to \{-1, 0, 1\} \\
dcmp &: (Double' \times Double') \to \{-1, 0, 1\}
\end{aligned}$$

where $Float' = Float \backslash \{\texttt{NaN} : Float\}$ and $Double' = Double \backslash \{\texttt{NaN} : Double\}$

The abstract type *Int* is considered ordered with respect to the $<$ operator. Similarly, the abstract type *PC* is considered ordered with respect to $<$. The abstract types *Long*, *Float*, and *Double* are only considered ordered with respect to the above comparison functions *lcmp*, *fcmp*, and *dcmp*, respectively.

The global, static environment containing class and interface declarations (i.e., the file system holding Java class files in a concrete implementation of the Java Virtual Machine) is represented by this partial function:

$$global : Id_c \to C$$

A number of standard Java classes are assumed to be declared in the global environment:

$$\begin{aligned} dom(global) \supseteq \{&\texttt{java.lang.Object}, \texttt{java.lang.Class}, \texttt{java.lang.String}, \\ &\texttt{java.lang.Throwable}, \texttt{java.lang.Error}, \texttt{java.lang.LinkageError}, \\ &\texttt{java.lang.NoClassDefFoundError}\} \end{aligned}$$

These abstract well-formedness predicates are used for classifying values of type C, *Env*, and *Heap*:

$$\begin{aligned} wf_c &: C \to \mathcal{B} \\ wf_e &: Env \to \mathcal{B} \\ wf_h &: Heap \to \mathcal{B} \end{aligned}$$

The predicate $wf_c(c)$ asserts that the static constraints, as described in Section 4.1 through 4.8.1 of *The Java Virtual Machine Specification*[2], hold for a single class declaration $c \in C$.

For any environment $e \in Env$ the wf_e predicate must satisfy

$$\begin{aligned} wf_e(e) \Leftrightarrow ((&\forall (c, sv) \in rng(e).wf_c(c)) \wedge \\ &(\forall id_1, id_2 \in dom(e).(id_1 \neq id_2 \Rightarrow (id_1 \notin supers(id_2, e) \vee id_2 \notin supers(id_1, e)))) \wedge \\ &(\forall id_c \in dom(e).(id_c \neq super(id_c, e) \ \wedge \ \texttt{java.lang.Object} \in supers(id_c, e)))) \wedge \\ &\ldots \end{aligned}$$

where $supers(id_c, e)$ is the set of superclasses of id_c according to the environment e; that is, any class or interface declaration in e must be well-formed, no class or interface may be declared to be its own superclass, and any class or interface, except `java.lang.Object` itself, must be declared to have class `java.lang.Object` as its superclass.

Note that the above is not considered an exhaustive definition of $wf_e(e)$; this predicate is inteded to represent further verification of properties of e, e.g. required relations between a class and its superclasses, and between a class and its superinterfaces, although this has not been specified here.

The utility functions *supers* and *super* are defined in the following section.

A string object may be tagged as being "interned"; the abstract predicate

$$interned : Obj \to \mathcal{B}$$

asserts that an object is an instance of class `java.lang.String`, and has been tagged as "interned".

For any heap $h \in Heap$ the wf_h predicate must satisfy

$$\begin{aligned} wf_h(h) \Leftrightarrow \forall r, r' \in dom(h).((&interned(h(r)) \wedge interned(h(r')) \wedge \\ &stringOf(h(r)) = stringOf(h(r'))) \ \Rightarrow \ r = r') \end{aligned}$$

where the abstract function

$$stringOf : Obj \rightarrow String$$

extracts the *String* value of a `java.lang.String` object. The *stringOf* function is assumed to be defined only for instances of class `java.lang.String`.

The following abstract function is used for creation of a `java.lang.String` object representing a given *String* value (string literal):

$$newString : (String \times Heap \times Env) \rightarrow (Ref \times Heap \times Env)$$

For any $v \in String, h \in Heap$, and $e \in Env$, the *newString* function must satisfy

$$\begin{aligned} newString(v,h,e) = (r,h',e') \Rightarrow (&h \subseteq h' \wedge e \subseteq e' \wedge \texttt{java.lang.String} \in dom(e') \wedge \\ &(wf_h(h) \Rightarrow wf_h(h')) \ \wedge \ (wf_e(e) \Rightarrow wf_e(e')) \wedge \\ &r \in dom(h') \wedge interned(h'(r))) \end{aligned}$$

If the specified environment e does not include the declaration of class `java.lang.String`, *newString* must first load that declaration from the global environment, and initialize the class; hence the modified environment e' must contain the declaration of `java.lang.String`.

If the specified heap h already contains a string object tagged as "interned", and having the specified *String* value, then *newString* must return a reference to that object; otherwise *newString* must create a new `java.lang.String` object, tagged as "interned", and return a reference to that object.

Assuming $\{\texttt{java.lang.Object}, \texttt{java.lang.String}\} \subseteq dom(global)$, and assuming the *Heap* operand h has unlimited size, *newString* is a total function.

The usual arithmetic operators are not defined for the abstract type PC; program counter values can only by manipulated by means of the following partial functions:

$$\begin{array}{lll} succ & : & TS \rightarrow PC \quad \text{where } succ(ts) = pc' \Rightarrow pc' \in valid_{pc}(ts) \\ jump & : & (TS \times Offset) \rightarrow PC \quad \text{where } jump(ts,\delta) = pc' \Rightarrow pc' \in valid_{pc}(ts) \end{array}$$

The function *succ* is only defined for a thread state $ts \in TS$ if it is valid to continue with the next instruction, at pc', in the current method of ts.

The function *jump* is only defined for a thread state $ts \in TS$ and an offset $\delta \in Offset$ if it is valid to continue with the instruction at pc', namely at offset δ from the current instruction, in the current method of ts.

Whether or not it is 'valid' to continue with a specific target instruction is defined in terms of the semantic function $valid_{pc}$ (cf. Section 2.4.3).

This abstract predicate asserts that two classes or interfaces belong to the same package:

$$samePackage : Id_c \times Id_c \rightarrow \mathcal{B}$$

Packages are not considered any further in this specification of the Java Virtual Machine semantics.

The following abstract function represents the (optional) mechanism of the Java Virtual Machine to invoke native methods, that is, methods implemented by platform-dependent code:

$$invokeNative : TS \rightarrow TS$$

Note that a native method is assumed to have unlimited access to the entire internal state of the Java Virtual Machine, and may change it in any conceivable way (cf. rule 57).

2.4.2 Utility Functions

In this section we define a number of utility functions to be used in the specification of the semantics of the Java Virtual Machine instructions. Unless otherwise stated, the utility functions are total.

The operator @ is used for appending two lists or sequences:

$$<x_1, x_2, x_3, \ldots, x_n>@<y_1, y_2, y_3, \ldots, y_m> \equiv <x_1, x_2, x_3, \ldots, x_n, y_1, y_2, y_3, \ldots, y_m>$$

The function *size* calculates the number of machine words used for storing a value of type *Oper*:

$$size(x : Oper) = \begin{cases} 1 & \text{if } x \in W \\ 2 & \text{if } x \in DW \end{cases}$$

Another variant of the *size* function calculates the total size of a list of *Oper* values. This is used for checking the size of an operand stack against the maximum stack size (measured in number of words):

$$size(<x_1, x_2, x_3, \ldots, x_n> : Oper^*) = \sum_{i=1}^{n} size(x_i)$$

Similarly, yet another variant of the *size* function calculates the total size of the *Oper* values stored in a set of local variables. This is used for checking the size of the local variables against the maximum size (measured in number of words):

$$size(l : Locals) = \sum_{i=1}^{n} size(l(k_i)) \qquad \text{where } dom(l) = \{k_0, k_1, k_2, \ldots, k_n\}$$

The following function removes the 'first half' of a local variable taking two local variable entries. This is used in connection with instructions storing a value into a local variable; if the preceding local variable entry contains the first half of a two-word value (occupying two consecutive local variable entries), then that entry must be removed (cf. rule 14 ff):

$$rmDW(l : Locals, j : \mathcal{N}_0) = \begin{cases} l_{j-1} & \text{if } j > 0 \ \wedge \ l(j-1) \in DW \\ l & \text{otherwise} \end{cases}$$

The function *if* selects one of two values of type PC, based on the specified boolean value. This is used for specifying the choice between two branches in branching instructions (cf. Section 2.5.10, rule 34 ff):

$$if(b : \mathcal{B}, pc, pc' : PC) = \begin{cases} pc & \text{if } b \\ pc' & \text{otherwise} \end{cases}$$

The following function calculates an offset value, based on the specified 'jump table' θ, key k, and default offset δ. This is used in connection with the `lookupswitch` and `tableswitch` instructions (cf. rule 36 ff):

$$switch(k : Int, \theta : Int \rightarrow Offset, \delta : Offset) = \begin{cases} \theta(k) & \text{if } k \in dom(\theta) \\ \delta & \text{otherwise} \end{cases}$$

The following function inserts the elements of the specified operand list into the first entries of a fresh set of local variables. This is used in connection with method invocation, for passing a number of stack operands as the first local variables of the invoked method (cf. rule 48 ff):

$$args(<a_n, a_{n-1}, a_{n-2}, \ldots, a_1, a_0> : Oper^*) = \{k_i \mapsto a_i \mid 0 \leq i \leq n\} : Locals$$
$$\text{where } k_i = size(<a_0, a_1, \ldots, a_{i-1}>) : Int$$

Note that although the result of the *size* function is a non-negative integer $n \in \mathcal{N}_0$, the value of each k_i is treated as an *Int*.

The function min_{pc} calculates the address (program counter) of the first instruction in a code array. This is used for specifying the initial program counter of a method in connection with method invocation (cf. rule 58 and 48 ff):

$$min_{pc}(c : Code) = pc : PC$$
$$\text{where } \forall pc' \in dom(c).pc \leq pc'$$
$$\text{undefined if } dom(c) = \emptyset$$

The function *supers* calculates the set of class names identifying the specified class itself and all of its superclasses:

$$supers(id_c : Id_c, e : Env) = \begin{cases} \{id_c\} & \text{if } id_c = \texttt{java.lang.Object} \\ \{id_c\} \cup ss & \text{if } super(id_c, e) = id_c' \wedge supers(id_c', e) = ss \\ \text{undefined} & \text{otherwise} \end{cases}$$

The *supers* function is undefined if any of the superclasses of the specified class are not in the environment e. Note that class `java.lang.Object` is assumed to have no direct superclass.

The function *super* returns the name of the direct superclass of the specified class, provided the class is not `java.lang.Object`, and has already been loaded into the specified environment:

$$super(id_c : Id_c, e : Env) = id_c' : Id_c$$
$$\text{where } e(id_c) = ((acc_c, id_c', is, fd, cv, md, mi, cp), sv)$$
$$\text{undefined if } id_c \notin dom(e) \ \vee \ id_c = \texttt{java.lang.Object}$$

The following function calculates the set of names of interfaces implemented by the specified class, provided these interfaces have already been loaded into the specified environment:

$$interfaces(id_c : Id_c, e : Env) =$$
$$\begin{cases} is \cup is'' & \text{if } id_c = \texttt{java.lang.Object} \wedge \\ & \quad e(id_c) = ((acc_c, id_c', is, fd, cv, md, mi, cp), sv) \wedge \\ & \quad interfaces(is, e) = is'' \\ is \cup is' \cup is'' & \text{if } id_c \neq \texttt{java.lang.Object} \wedge \\ & \quad e(id_c) = ((acc_c, id_c', is, fd, cv, md, mi, cp), sv) \wedge \\ & \quad interfaces(id_c', e) = is' \wedge interfaces(is, e) = is'' \\ \text{undefined} & \text{otherwise} \end{cases}$$

Another variant of the *interfaces* function calculates the set of names of interfaces that the specified set of classes or interfaces implement or extend, provided all of these interfaces have already been loaded into the specified environment:

$$interfaces(\{id_1, id_2, id_3, \ldots, id_n\} : \mathcal{P}(Id_c), e : Env) = \bigcup_{i=1}^{n} interfaces(id_i, e)$$
$$\text{undefined if } \exists 1 \leq i \leq n.(id_i, e) \notin dom(interfaces)$$

The following function prepares the set of instance fields declared by the specified class and all of its superclasses (if any). This is used in connection with the class instantiation instruction `new` (cf. rule 37):

$$\mathit{fields}(id_c : Id_c, e : Env) =$$
$$\begin{cases} \mathit{fields}(id_c, fd) & \text{if } id_c = \texttt{java.lang.Object} \wedge \\ & \quad e(id_c) = ((acc_c, id_c', is, fd, cv, md, mi, cp), sv) \\ \mathit{fields}(id_c, fd) \cup fs & \text{if } id_c \neq \texttt{java.lang.Object} \wedge \\ & \quad e(id_c) = ((acc_c, id_c', is, fd, cv, md, mi, cp), sv) \wedge \\ & \quad \mathit{fields}(id_c', e) = fs \\ \text{undefined} & \text{otherwise} \end{cases}$$

The following variant of the *fields* function prepares the set of instance fields declared in a single class; each instance field is initialized to the default value for its type:

$$\mathit{fields}(id_c : Id_c, fd : FD) = \{(id_c, id_f) \mapsto \mathit{default}(d) \mid fd(id_f) = (acc_f, d) \wedge \texttt{static} \notin acc_f\}$$

This function calculates a default value for a simple or non-simple type, based on the specified descriptor for the type:

$$\mathit{default}(d : Desc) = \begin{cases} 0 : Int & \text{if } d \in \{\texttt{boolean}, \texttt{char}, \texttt{byte}, \texttt{short}, \texttt{int}\} \\ 0 : Long & \text{if } d = \texttt{long} \\ 0 : Float & \text{if } d = \texttt{float} \\ 0 : Double & \text{if } d = \texttt{double} \\ \texttt{null} & \text{if } d \notin Simple \end{cases}$$

Note that the *default* function returns the value $0 : Int$ for descriptors of the 'small' Java types `boolean`, `char`, `byte`, and `short`. For `char`, `byte`, and `short` descriptors, this corresponds to the default value for those types, as defined in *The Java Virtual Machine Specification*[2, p. 12], extended to a value of type *Int*. For a `boolean` descriptor, the result returned by the *default* function corresponds to the assumed encoding of the default `boolean` value, `false`, as the *Int* value 0 (cf. Section 3.3.4).

The function *lookup* searches for the declaration and implementation of the specified method in the specified class; if a method with the specified signature is not declared in the specified class, the superclass of that class is then searched (and so on, recursively). This is used in connection with the method invocation instructions (cf. rule 49 ff):

$$\mathit{lookup}(sig : Sig, id_c : Id_c, e : Env) =$$
$$\begin{cases} (id_c, acc_m, d, n_l, code') & \text{if } e(id_c) = ((acc_c, id_c', is, fd, cv, md, mi, cp), sv) \wedge \\ & \quad md(sig) = (acc_m, d, excs) \wedge \\ & \quad mi(sig) = (n_s, n_l, code', hdls') \\ (id_c'', acc_m, d, n_l, code') & \text{if } e(id_c) = ((acc_c, id_c', is, fd, cv, md, mi, cp), sv) \wedge \\ & \quad sig \notin dom(md) \ \wedge \ id_c' \in Id_c \wedge \\ & \quad \mathit{lookup}(sig, id_c', e) = (id_c'', acc_m, d, n_l, code') \\ \text{undefined} & \text{otherwise} \end{cases}$$

The following function returns a value of 1 or 0 (of type *Int*) depending on whether or not the specified reference refers to an instance of the specified type. This is used in connection with the type-checking instruction `instanceof` (cf. rule 61):

$$\mathit{instOf}_i(d : (Array \cup Id_c), r : Ref_0, h : Heap, e : Env) =$$
$$\begin{cases} 1 : Int & \text{if } r \in dom(h) \ \wedge \ \mathit{instOf}(d, h(r), e) \\ 0 : Int & \text{otherwise} \end{cases}$$

The *instOf* predicate is defined in Section 2.4.4.

The function *arrayDesc* builds an array type descriptor based on the specified component type descriptor. This is used in connection with the `anewarray` and `newarray` instructions (cf. rule 42 ff):

$$arrayDesc(d : Desc) = \begin{cases} (n+1,t) : Array & \text{if } d = (n,t) : Array \\ (1,d) : Array & \text{otherwise} \end{cases}$$

The function *compDesc* returns the component type descriptor from the specified array type descriptor; in other words, this is the inverse of the *arrayDesc* function. The *compDesc* function is used in the following definitions, and in connection with the `aastore` instruction (cf. rule 23):

$$compDesc((n,t) : Array) = \begin{cases} t : (Simple \cup Id_c) & \text{if } n = 1 \\ (n-1,t) : Array & \text{otherwise} \end{cases}$$

The following function creates one or more dimensions of an array object of the specified array type. This is used in connection with the `multianewarray` instruction (cf. rule 43):

$$newArray(d : Array, ks : Int^*, h : Heap) = \begin{cases} singleArray(d,k,h) & \text{if } ks = <k> \\ (r,h') & \text{if } ks = <k,k'>@kr \ \wedge \ multiArray(d,ks,h) = (r,h') \\ \text{undefined} & \text{otherwise} \end{cases}$$

Note that not all dimensions of a multi-dimensional array have to be created at once; if the operand *ks* specifies that fewer dimensions than the number of dimensions in the operand *d* are to be created, then all of the array components in the 'lowest' dimension will be initialized to the special value `null`.

The function *singleArray* creates (a single dimension of) an array object of the specified array type:

$$\begin{aligned} &singleArray(d : Array, k : Int, h : Heap) = (r : Ref, h' : Heap) \\ &\quad \text{where } default(compDesc(d)) = v : V \\ &\qquad \{(k' : Int) \mapsto v \mid 0 \le k' < k\} = av \\ &\qquad (d,k,av) = o : Obj_a \\ &\qquad r \in Ref \backslash dom(h) \\ &\qquad h + \{r \mapsto o\} = h' \end{aligned}$$

Note that values of type *Int* are considered ordered with respect to $<$.

The function *multiArray* creates two or more dimensions of an array of the specified array type. The first dimension of the resulting array is an array object whose components are initialized as references to separate array objects of the next dimension:

$$\begin{aligned} &multiArray(d : Array, (k : Int) :: kr, h_0 : Heap) = (r : Ref, h' : Heap) \\ &\quad \text{where } compDesc(d) = d' : Array \\ &\qquad newArray(d', kr, h_i) = (r_i : Ref, h_{i+1} : Heap) \text{ for } 0 \le i < k \\ &\qquad \{(k' : Int) \mapsto r_{k'} \mid 0 \le k' < k\} = av \\ &\qquad (d,k,av) = o : Obj_a \\ &\qquad r \in Ref \backslash dom(h_0) \\ &\qquad h_k + \{r \mapsto o\} = h' \\ &\quad \text{undefined if } compDesc(d) \notin Array \ \vee \ \exists 0 \le i < k.(d', kr, h_i) \notin dom(newArray) \end{aligned}$$

Note that although the values of k and k' are of type Int, these values are also treated as non-negative integers in some places.

The following function changes the 'state' of an object referred to by the specified reference from being uninitialized to initialized, provided the specified method is the constructor of class `java.lang.Object`. This is used in connection with the method invocation instruction `invokespecial` for permitting future access to members of the (as of then) initialized object (cf. rule 52):

$$initObj(id_c : Id_c, sig : Sig, r : Ref, h : Heap) = \begin{cases} h + \{r \mapsto (id_c', iv) : Obj_c\} & \text{if } id_c = \texttt{java.lang.Object} \ \wedge \ sig = (\texttt{<init>}, <>) \ \wedge \\ & \quad h(r) = (id_c', iv) : Obj_u \\ h & \text{otherwise} \end{cases}$$

The function *loadClass* loads the specified class or interface and all of its superclasses which have not yet been loaded, prepares these, and returns a list of their initializer methods. This is used for initialization of classes and interfaces (cf. rule 58):

$$loadClass(id_c : Id_c, h : Heap, e : Env) = \begin{cases} (<>, h, e) & \text{if } id_c \in dom(e) \\ (cs, h', e') & \text{if } id_c \notin dom(e) \ \wedge \ global(id_c) = (acc_c, id_c', is, fd, cv, md, mi, cp) = c \ \wedge \\ & \quad id_c = \texttt{java.lang.Object} \ \wedge \\ & \quad prepare(c, h, e) = (h', e') \ \wedge \ clInit(id_c, mi, <>) = cs \\ (cs', h'', e'') & \text{if } id_c \notin dom(e) \ \wedge \ global(id_c) = (acc_c, id_c', is, fd, cv, md, mi, cp) = c \ \wedge \\ & \quad id_c' \in Id_c \ \wedge \ loadClass(id_c', h, e) = (cs, h', e') \ \wedge \\ & \quad prepare(c, h', e') = (h'', e'') \ \wedge \ clInit(id_c, mi, cs) = cs' \\ \text{undefined} & \text{otherwise} \end{cases}$$

The function *prepare* initializes the set of static fields declared in the specified class declaration. Static fields for which the class declaration defines constant values are initialized to the corresponding constant values; other static fields in the class are initialized to the default values corresponding to their types:

$$\begin{aligned} &prepare(c : C, h : Heap, e : Env) = (h', e'') \\ &\quad \text{where } c = (acc_c, id_c', is, fd, cv, md, mi, cp) \\ &\qquad \{id_f \mapsto default(d) \mid fd(id_f) = (acc_f, d) \wedge \texttt{static} \in acc_f\} = sv \\ &\qquad constValues(cv, h, e) = (sv', h', e') \\ &\qquad e' + \{id_c \mapsto (c, sv + sv')\} = e'' \end{aligned}$$

The following function initializes a set of static fields from a set of constant values:

$$\begin{aligned} &constValues(cv : CV, h_0 : Heap, e_0 : Env) = (sv : SV, h_n : Heap, e_n : Env) \\ &\quad \text{where } dom(cv) = \{id_1, id_2, id_3, \ldots, id_n\} \\ &\qquad constVal(cv(id_i), h_{i-1}, e_{i-1}) = (v_i, h_i, e_i) \text{ for } 1 \leq i \leq n \\ &\qquad \{id_i \mapsto v_i \mid 1 \leq i \leq n\} = sv \end{aligned}$$

The function *constVal* converts a single constant value to an object or to a value of simple type, suitable for initialization of a static field (see above). This is also used in connection with instructions loading constant pool values onto the operand stack (cf. rule 3 ff):

$$constVal(v : Const_v, h : Heap, e : Env) = \begin{cases} newString(v, h, e) & \text{if } v \in String \\ (v, h, e) & \text{otherwise} \end{cases}$$

The function *clInit* is used by the *loadClass* function to build a list of initializer methods for the classes being loaded. If the specified class declares an initializer method, then the name of the class and the code array for that method is appended to the end of the specified list of initializers; otherwise the list is returned unchanged:

$$clInit(id_c : Id_c, mi : MI, cs : (Id_c \times Code)^*) =$$
$$\begin{cases} cs@<(id_c, code')> & \text{if } mi(\texttt{<clinit>}, <>) = (n_s, n_l, code', hdls') \\ cs & \text{if } (\texttt{<clinit>}, <>) \notin dom(mi) \end{cases}$$

The following function throws an exception object (referred to by the specified reference), and transfers control to the 'closest' exception handler in the specified thread state. This is used in connection with the `athrow` instruction (cf. rule 59):

$$throw(r : Ref, ts : TS) =$$
$$\begin{cases} ((pc', <r>, l, m) :: fr, h, e) : TS & \text{if } fs = (pc, s, l, m) :: fr \ \wedge \ h(r) = (id_c', iv) : Obj_c \ \wedge \\ & \quad supers(id_c', e) = ss \ \wedge \ hdls(ts) = hdls' \ \wedge \\ & \quad hdls'(pc) = hdls'' \ \wedge \ hdls''(ss) = pc' \ \wedge \\ & \quad pc' \in valid_{pc}(ts) \\ ts' : TS & \text{if } fs = f :: fr \ \wedge \ h(r) = (id_c', iv) : Obj_c \ \wedge \\ & \quad supers(id_c', e) = ss \ \wedge \ hdls(ts) = hdls' \ \wedge \\ & \quad (hdls'(pc) = hdls'' \Rightarrow ss \notin dom(hdls'')) \ \wedge \\ & \quad throw(r, (fr, h, e)) = ts' \\ \text{undefined} & \text{otherwise} \end{cases}$$
$$\text{where } ts = (fs, h, e)$$

It is first checked whether the current method of the thread state defines any exception handler covering the current program counter of the method. If so, it is checked if any such handler is declared to handle the class of the thrown exception or one of its superclasses; if so, the program counter is set to the start of the handler, and the contents of the operand stack is replaced by the reference to the thrown exception object.

If the current method of the thread state does not define one or more handlers covering the current program counter, or if none of these handles the thrown exception, then the current frame is discarded, and the exception is 'rethrown' in the previous frame on the frame stack, corresponding to the invoker of the current method. If the exception is also not handled within the frame of the invoker, then the exception is rethrown to the invoker of the invoker, etc.

Note that the *throw* function is undefined if the thrown exception is not handled within any of the frames on the frame stack; in other words, the semantics of unhandled exceptions is not specified.

The $hdls$ and $valid_{pc}$ thread state query functions are defined in the following section.

2.4.3 Thread State Queries

In this section we define a set of semantic functions for extracting components of a thread state, and for calculating information derived therefrom.

The following functions extract the program counter, operand stack, local variables, and method identifier, respectively, from the current frame of the specified thread state:

$$\left.\begin{array}{rcl} cur_{pc}((fs, h, e) : TS) & = & pc : PC \\ cur_s((fs, h, e) : TS) & = & s : Oper^* \\ cur_l((fs, h, e) : TS) & = & l : Locals \\ cur_m((fs, h, e) : TS) & = & m : (Id_c \times Sig) \end{array}\right\} \quad \begin{array}{l} \text{where } fs = (pc, s, l, m) :: fr \\ \text{undefined if } fs = <> \end{array}$$

These functions extract the heap and environment components, respectively, from the specified thread state:

$$\begin{aligned} heap((fs, h, e) : TS) &= h : Heap \\ env((fs, h, e) : TS) &= e : Env \end{aligned}$$

The function *pool* returns the constant pool of the class of the current method in the specified thread state:

$$\begin{aligned} &pool((fs, h, e) : TS) = cp : CP \\ &\quad \text{where } fs = (pc, s, l, (id_c, sig)) :: fr \\ &\qquad\quad e(id_c) = ((acc_c, id_c{}', is, fd, cv, md, mi, cp), sv) \\ &\quad \text{undefined if } fs =<> \ \vee \ id_c \notin dom(e) \end{aligned}$$

The following function returns the implementation of the current method in the specified thread state:

$$\begin{aligned} &impl((fs, h, e) : TS) = mi(sig) : Impl \\ &\quad \text{where } fs = (pc, s, l, (id_c, sig)) :: fr \\ &\qquad\quad e(id_c) = ((acc_c, id_c{}', is, fd, cv, md, mi, cp), sv) \\ &\quad \text{undefined if } fs =<> \ \vee \ id_c \notin dom(e) \ \vee \ sig \notin dom(mi) \end{aligned}$$

The following functions extract the operand stack limit, local variable limit, code array, and exception handlers, respectively, from the implementation of the current method in the specified thread state:

$$\left.\begin{aligned} max_s(ts : TS) &= n_s : \mathcal{N}_0 \\ max_l(ts : TS) &= n_l : \mathcal{N}_0 \\ code(ts : TS) &= code' : Code \\ hdls(ts : TS) &= hdls' : H \end{aligned}\right\} \quad \begin{aligned} &\text{where } impl(ts) = (n_s, n_l, code', hdls') \\ &\text{undefined if } ts \notin dom(impl) \end{aligned}$$

This function returns the current instruction of the current method in the specified thread state:

$$\begin{aligned} &instr(ts : TS) = code'(pc) : Instr \\ &\quad \text{where } cur_{pc}(ts) = pc \\ &\qquad\quad code(ts) = code' \\ &\quad \text{undefined if } ts \notin dom(cur_{pc}) \cap dom(code) \ \vee \ pc \notin dom(code') \end{aligned}$$

The function $valid_{pc}$ returns the set of program counter values (instruction addresses) that it is valid to continue with, given the contents of the operand stack and local variables in the current frame of the specified thread state:

$$\begin{aligned} &valid_{pc}(ts : TS) = \\ &\quad \{pc' \in dom(code(ts)) \mid (pc \le pc' \wedge pc' \notin dom(hdls(ts))) \vee \\ &\qquad\qquad\qquad\qquad (pc \le pc' \wedge allInit_l(ts)) \vee \\ &\qquad\qquad\qquad\qquad (allInit_l(ts) \wedge allInit_s(ts))\} \\ &\quad \text{where } cur_{pc}(ts) = pc : PC \\ &\quad \text{undefined if } ts \notin dom(cur_{pc}) \cap dom(hdls) \end{aligned}$$

The $valid_{pc}$ function is used for enforcing these restrictions concerning uninitialized objects:

> *There must never be an uninitialized class instance on the operand stack or in a local variable when any backwards branch is taken. There must never be an uninitialized class instance in a local variable in code protected by an exception handler ...*
>
> (*The Java Virtual Machine Specification*[2, p. 122])

The predicates $allInit_l$ and $allInit_s$ are defined in the following section.

2.4.4 Predicates

In this section we define a number of predicates to be used in the premises of the rules for each of the Java Virtual Machine instructions.

The predicate *initialized* asserts that the specified operand v is not a reference to an uninitialized object ($o \in Obj_u$) in the specified heap h:

$$initialized(v : Oper, h : Heap) \;\Leftrightarrow\; (v \in dom(h) \Rightarrow h(v) \in Obj_c \cup Obj_a)$$

The predicate $allInit_s$ asserts that the current operand stack of the specified thread state ts does not contain any references to uninitialized objects in the heap of ts:

$$allInit_s(ts : TS) \;\Leftrightarrow\; \forall 1 \leq k \leq n.initialized(s_k, heap(ts))$$
$$\text{where } cur_s(ts) = <s_1, s_2, s_3, \ldots, s_n> : Oper^*$$

Similarly, the predicate $allInit_l$ asserts that none of the local variables in the current frame of the specified thread state ts refer to uninitialized objects in the heap of ts:

$$allInit_l(ts : TS) \;\Leftrightarrow\; \forall k \in dom(l).initialized(l(k), heap(ts))$$
$$\text{where } cur_l(ts) = l : Locals$$

The predicate *compatVal* asserts that the type of the specified operand v is assignment compatible with the type of the specified descriptor d:

$$\begin{aligned} &compatVal(d : Desc, v : Oper, h : Heap, e : Env) \;\Leftrightarrow \\ &\quad ((v \in Int \wedge d \in \{\texttt{boolean}, \texttt{char}, \texttt{byte}, \texttt{short}, \texttt{int}\}) \vee \\ &\quad (v \in Long \wedge d = \texttt{long}) \vee \\ &\quad (v \in Float \wedge d = \texttt{float}) \vee \\ &\quad (v \in Double \wedge d = \texttt{double}) \vee \\ &\quad (v = \texttt{null} \wedge d \notin Simple) \vee \\ &\quad (v \in dom(h) \wedge instOf(d, h(v), e))) \end{aligned}$$

The predicate *instOf* asserts that the specified object o is an instance of the class or array type of the specified descriptor d:

$$\begin{aligned} &instOf(d : (Array \cup Id_c), o : Obj, e : Env) \;\Leftrightarrow \\ &\quad ((o = (id_c, iv) : Obj_c \wedge compat(d, id_c, e)) \;\vee (o = (d', k, av) : Array \wedge compat(d, d', e))) \end{aligned}$$

The predicate *compat* asserts that the type of the specified source descriptor d_s is assignment compatible with the type of the specified target descriptor d_t:

$$\begin{aligned} &compat(d_t, d_s : Desc, e : Env) \;\Leftrightarrow \\ &\quad ((d_s \in Id_c \wedge d_t \in supers(d_s, e) \cup interfaces(d_s, e)) \vee \\ &\quad (d_s \in Array \wedge d_t : Id_c \in \{\texttt{java.lang.Object}, \texttt{java.lang.Cloneable}\}) \vee \\ &\quad (d_s \in Array \wedge d_t \in Array \wedge compatArray(d_t, d_s, e))) \end{aligned}$$

The predicate *compatArray* asserts that the array type of the specified source descriptor d_s is assignment compatible with the array type of the specified target descriptor d_t:

$$\begin{aligned} &compatArray(d_t, d_s : Array, e : Env) \;\Leftrightarrow \\ &\quad ((d'_t \in Simple \wedge d'_t = d'_s) \vee \\ &\quad (d'_t \notin Simple \wedge d'_s \notin Simple \wedge compat(d'_t, d'_s, e))) \\ &\quad \text{where } compDesc(d_t) = d'_t \text{ and } compDesc(d_s) = d'_s \end{aligned}$$

The predicate *access* asserts that the class identified by id_s may access (members of) the class or interface identified by id_t, having access modifiers acc_t:

$$access(id_t : Id_c, acc_t : \mathcal{P}(Acc_c), id_s : Id_c) \Leftrightarrow$$
$$(id_t \neq id_s \Rightarrow (samePackage(id_t, id_s) \vee \texttt{public} \in acc_t))$$

Another variant of the predicate *access* asserts that class id_s may access class or interface id_t, provided the access modifiers of id_t can be retrieved via the specified environment e and access is granted according to the above predicate:

$$access(id_t, id_s : Id_c, e : Env) \Leftrightarrow$$
$$(e(id_t) = ((acc_c, id_c{}', is, fd, cv, md, mi, cp), sv) \wedge access(id_t, acc_c, id_s))$$

2.5 Semantics of Instructions

The following sections describe the semantics of the instructions of the Java Virtual Machine. The semantics is described in terms of a set of transition rules, defining under which circumstances the state of the Java Virtual Machine will change, and how.

The following bindings are assumed to be part of each of the transition rules:

$$\begin{aligned} ts &= (f :: fr, h : Heap, e : Env) : TS \\ f &= (pc : PC, s : Oper^*, l : Locals, m) : Frame \\ m &= (id_c : Id_c, sig : Sig) \end{aligned}$$

The symbol ts is considered the name of the 'current' thread state, that is, the state of the Java Virtual Machine before the instruction in question is executed.

The symbol f is the name of the current frame in the thread state ts; for all of the transition rules it is assumed that the frame stack is not empty, i.e., that there is a current frame. See Sections 2.6 and 2.7 for a discussion of abrupt and normal program termination.

The symbol m is the name of the method identifier of the current frame f; the method identifier consists of the name of the class implementing the current method, and the signature of the method.

2.5.1 Instruction Set

This is the complete instruction set of the Java Virtual Machine:

Instr = { `aaload`, `aastore`, `aconst_null`, `aload` j, `aload_<n>`, `anewarray` i, `areturn`, `arraylength`, `astore` j, `astore_<n>`, `athrow`, `baload`, `bastore`, `bipush` k, `caload`, `castore`, `checkcast` i, `d2f`, `d2i`, `d2l`, `dadd`, `daload`, `dastore`, `dcmpg`, `dcmpl`, `dconst_<d>`, `ddiv`, `dload` j, `dload_<n>`, `dmul`, `dneg`, `drem`, `dreturn`, `dstore` j, `dstore_<n>`, `dsub`, `dup`, `dup_x1`, `dup_x2`, `dup2`, `dup2_x1`, `dup2_x2`, `f2d`, `f2i`, `f2l`, `fadd`, `faload`, `fastore`, `fcmpg`, `fcmpl`, `fconst_<f>`, `fdiv`, `fload` j, `fload_<n>`, `fmul`, `fneg`, `frem`, `freturn`, `fstore` j, `fstore_<n>`, `fsub`, `getfield` i, `getstatic` i, `goto` δ, `goto_w` δ, `i2b`, `i2c`, `i2d`, `i2f`, `i2l`, `i2s`, `iadd`, `iaload`, `iand`, `iastore`, `iconst_<i>`, `idiv`, `if_acmpeq` δ, `if_acmpne` δ, `if_icmp<cond>` δ, `if<cond>` δ, `ifnonnull` δ, `ifnull` δ, `iinc` j k, `iload` j, `iload_<n>`, `imul`, `ineg`, `instanceof` i, `invokeinterface` i n, `invokespecial` i, `invokestatic` i, `invokevirtual` i, `ior`, `irem`, `ireturn`, `ishl`, `ishr`, `istore` j, `istore_<n>`, `isub`, `iushr`, `ixor`, `jsr` δ, `jsr_w` δ, `l2d`, `l2f`, `l2i`, `ladd`, `laload`, `land`, `lastore`, `lcmp`, `lconst_<l>`, `ldc` i, `ldc_w` i, `ldc2_w` i, `ldiv`, `lload` j, `lload_<n>`, `lmul`, `lneg`, `lookupswitch` δ θ, `lor`, `lrem`, `lreturn`, `lshl`, `lshr`, `lstore` j, `lstore_<n>`, `lsub`, `lushr`, `lxor`, `monitorenter`, `monitorexit`, `multianewarray` i n, `new` i, `newarray` t, `nop`, `pop`, `pop2`, `putfield` i, `putstatic` i, `ret` j, `return`, `saload`, `sastore`, `sipush` k, `swap`, `tableswitch` δ k θ, `wide aload` j, `wide astore` j, `wide dload` j, `wide dstore` j, `wide fload` j, `wide fstore` j, `wide iinc` j k, `wide iload` j, `wide istore` j, `wide lload` j, `wide lstore` j, `wide ret` j }

where $i \in$ *Index*; $j \in \mathcal{N}_0$; $k \in$ *Int*; $n \in \mathcal{N}$; $\delta \in$ *Offset*; $\theta \in$ *Int* $\rightarrow$ *Offset*; $t \in$ *Simple*

The patterns `<n>`, `<d>`, `<f>`, `<i>`, `<cond>`, and `<l>` are used in some of the instruction mnemonics to specify 'families' of instructions:

`<n>` $\in$ {`0`, `1`, `2`, `3`}
`<d>` $\in$ {`0`, `1`}
`<f>` $\in$ {`0`, `1`, `2`}
`<i>` $\in$ {`m1`, `0`, `1`, `2`, `3`, `4`, `5`}
`<cond>` $\in$ {`eq`, `ne`, `lt`, `ge`, `gt`, `le`}
`<l>` $\in$ {`0`, `1`}

For example, the instruction set includes four different versions of `aload` with implicit (i.e., embedded) immediate operands: `aload_0`, `aload_1`, `aload_2`, and `aload_3`. Similarly for `astore_<n>`, `dconst_<d>`, `dload_<n>`, etc. Note that the instruction `iconst_m1` has the implicit immediate operand -1.

At the abstract level of this specification, the semantics of the `_w` instruction variants is the same as for the equivalent instructions without the `_w` suffix. For example, the semantics of the `goto_w` δ instruction is the same as that of the `goto` δ instruction. These two instructions differ only in the concrete representation of their immediate operand δ: `goto_w` takes a four-byte immediate operand, whereas `goto` takes a two-byte immediate operand.

Similarly, the `wide` instructions have the same semantics as the equivalent instructions without the `wide` prefix. For example, the semantics of the `wide ret` instruction (taking a two-byte immediate operand) is the same as that of the `ret` instruction (taking a one-byte immediate operand).

The `_w` instruction variants and the `wide` instructions are not specified explicitly in the following sections.

2.5.2 Immediate Operands

The instruction set of the Java Virtual Machine includes a number of instructions for pushing an immediate operand onto the operand stack, e.g.

$$\frac{\begin{array}{l} \mathit{instr}(ts) = \texttt{bipush}\ k \\ \mathit{size}(s) + \mathit{size}(k) \leq \mathit{max}_s(ts) \\ \mathit{succ}(ts) = pc' \end{array}}{ts \Rightarrow ((pc', k :: s, l, m) :: fr, h, e)} \quad (1)$$

The `bipush` instruction takes an immediate (`byte`) operand $k \in \mathit{Int}$.

If the operand stack does not thereby overflow, and if the address of the next instruction (pc') is in the set of valid target addresses for the thread state ts, then k is pushed onto the operand stack, and execution continues with the next instruction.

In the description of the following transition rules, the assumption that the operand stack does not overflow, and that the address of the next instruction is in the set of valid target addresses for the thread state ts, is referred to as the "standard assumptions".

The semantics of the instruction `sipush` is the same as that of the `bipush` instruction above. The only difference between these two instruction lies in the concrete representation of their immediate operands: `bipush` takes a one-byte operand, whereas `sipush` takes a two-byte (`short`) operand.

The semantics of the instructions with implicit immediate operands is similar to that of the `bipush` instruction above, e.g.

$$\frac{\begin{array}{l} \mathit{instr}(ts) = \texttt{lconst\_0} \\ \mathit{size}(s) + \mathit{size}(0 : \mathit{Long}) \leq \mathit{max}_s(ts) \\ \mathit{succ}(ts) = pc' \end{array}}{ts \Rightarrow ((pc', (0 : \mathit{Long}) :: s, l, m) :: fr, h, e)} \quad (2)$$

The immediate operand of the `lconst_0` instruction is encoded directly into the op-code of the instruction; this implicit operand (0 of type *Long*) is pushed onto the operand stack, and execution continues with the next instruction in the current method (under the standard assumptions).

Similarly, these instruction push their implicit immediate operands onto the operand stack:

- `lconst_1` pushes the value $1 : \mathit{Long}$
- `iconst_<i>` pushes the value -1, 0, 1, 2, 3, 4, or 5 of type *Int*
- `fconst_<f>` pushes the value 0.0, 1.0, or 2.0 of type *Float*
- `dconst_<d>` pushes the value 0.0 or 1.0 of type *Double*
- `aconst_null` pushes the special value `null`

2.5.3 Constant Pool Values

The instructions described in the previous section can only be used for pushing constant values of very limited subsets of the simple types. For the purpose of pushing other constant values of simple types, and for pushing references to constant strings (string literals), the instruction

set of the Java Virtual Machine also features instructions for loading a value from the constant pool, e.g.

$$\frac{\begin{array}{l} \mathit{instr}(ts) = \texttt{ldc}\ i \\ \mathit{pool}(ts)(i) = k : (\mathit{Int} \cup \mathit{Float}) \\ \mathit{size}(s) + \mathit{size}(k) \leq \mathit{max}_s(ts) \\ \mathit{succ}(ts) = pc' \end{array}}{ts \Rightarrow ((pc', k :: s, l, m) :: fr, h, e)} \quad (3)$$

The `ldc` instruction takes an immediate operand $i \in \mathit{Index}$. The constant pool of the class implementing the current method must include a value k, of type *Int* or *Float*, at index i. This value is pushed onto the operand stack, and execution continues with the next instruction in the current method (under the standard assumptions).

The semantics of the instruction `ldc2_w` is the same as that of the `ldc` instruction above, except that the constant pool value k must be of type *Long* or *Double*. Note that the instruction set of the Java Virtual Machine does not include an `ldc2` instruction[2].

The `ldc` instruction can also be used for loading a string literal from the constant pool:

$$\frac{\begin{array}{l} \mathit{instr}(ts) = \texttt{ldc}\ i \\ \mathit{pool}(ts)(i) = k : \mathit{String} \\ \mathit{newString}(k, h, e) = (r, h', e') \\ \mathit{size}(s) + \mathit{size}(r) \leq \mathit{max}_s(ts) \\ \mathit{succ}(ts) = pc' \end{array}}{ts \Rightarrow ((pc', r :: s, l, m) :: fr, h', e')} \quad (4)$$

For this rule to apply, the constant pool value k, at index i, must be of type *String*.

The *newString* function is used for creating an object of class `java.lang.String` representing the string literal k; this results in a reference r to the string object, a (possibly) extended heap h', and a (possibly) extended environment e'.

The reference r is pushed onto the operand stack, and execution continues with the next instruction (under the standard assumptions), now using the possibly modified heap h' and environment e'.

Note that *newString* is an abstract semantic function (see p. 10); it is thus not specified how the resulting `java.lang.String` instance was created (cf. Section 3.3.2).

2.5.4 Stack Manipulation

The instruction set of the Java Virtual Machine includes a few instructions for directly manipulating the operand stack in the current frame, e.g.

$$\frac{\begin{array}{l} \mathit{instr}(ts) = \texttt{dup} \\ s = (v : W) :: sr \\ \mathit{size}(s) + \mathit{size}(v) \leq \mathit{max}_s(ts) \\ \mathit{succ}(ts) = pc' \end{array}}{ts \Rightarrow ((pc', v :: v :: sr, l, m) :: fr, h, e)} \quad (5)$$

The `dup` instruction duplicates the topmost stack operand v, provided it is of type W (and under the standard assumptions).

[2]The concrete representation of the `ldc2_w` instruction takes a two-byte immediate operand i; there is no instruction taking a one-byte immediate operand for loading a constant pool value of type *Long* or *Double*.

The semantics of the instruction `dup2` is the same as that of the `dup` instruction above, except that the topmost stack operand v must be of type DW.

The `dup2` instruction can also be used if the two topmost stack operands v and v' are of type W:

$$\frac{\begin{array}{l} instr(ts) = \texttt{dup2} \\ s = vs@sr \\ vs = <v, v'> : W^* \\ size(s) + size(vs) \leq max_s(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', vs@vs@sr, l, m) :: fr, h, e)} \quad (6)$$

Again, the semantics of the `dup2` instruction used on two stack operands of type W is very similar to that of the `dup` instruction above (rule 5).

The semantics of the instruction `dup_x1` is also similar to that of the `dup` instruction above, except that it inserts a copy of the topmost stack operand v 'below' the two topmost stack operands, provided both of these are of type W:

$$\frac{\begin{array}{l} instr(ts) = \texttt{dup\_x1} \\ s = vs@sr \\ vs = <v, v'> : W^* \\ size(s) + size(v) \leq max_s(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', vs@(v :: sr), l, m) :: fr, h, e)} \quad (7)$$

The semantics of the instruction `dup_x2` is the same as that of the `dup_x1` instruction above (rule 7), except that the second-topmost stack operand v' must be of type DW. The semantics of the instruction `dup2_x1` is also the same as that of the `dup_x1` instruction, except that the topmost stack operand v must be of type DW. Furthermore, the semantics of the instruction `dup2_x2` is the same as that of the `dup_x1` instruction, except that both of the topmost stack operands v and v' must be of type DW.

The `dup_x2` instruction can also be used for inserting a copy of the topmost stack operand v below the three topmost stack operands, provided all of these are of type W:

$$\frac{\begin{array}{l} instr(ts) = \texttt{dup\_x2} \\ s = vs@sr \\ vs = <v, v', v''> : W^* \\ size(s) + size(v) \leq max_s(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', vs@(v :: sr), l, m) :: fr, h, e)} \quad (8)$$

Similarly, the `dup2_x2` instruction can be used if the topmost stack operand v is of type DW, and the next two stack operands are of type W. The semantics of the `dup2_x2` instruction, when used in this way, is similar to that of the `dup_x2` instruction above (rule 8).

The `dup2_x1` instruction can also be used for inserting a copy of the two topmost stack operands v and v' below the three topmost operands on the stack, provided all of these are of

type W:

$$\frac{\begin{array}{l} instr(ts) = \texttt{dup2\_x1} \\ s = vs@sr \\ vs = <v, v', v''> : W^* \\ size(s) + size(v) + size(v') \leq max_s(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', vs@(v :: v' :: sr), l, m) :: fr, h, e)} \quad (9)$$

Similarly, the `dup2_x2` instruction can be used if the two topmost stack operands v and v' are of type W, and the third-topmost stack operand v'' is of type DW. The semantics of the `dup2_x2` instruction, when used in this way, is similar to that of the `dup2_x1` instruction above (rule 9).

Furthermore, the `dup2_x2` instruction can be used for inserting a copy of the two topmost stack operands v and v' below the four topmost operands on the stack, provided these are all of type W:

$$\frac{\begin{array}{l} instr(ts) = \texttt{dup2\_x2} \\ s = vs@sr \\ vs = <v, v', v'', v'''> : W^* \\ size(s) + size(v) + size(v') \leq max_s(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', vs@(v :: v' :: sr), l, m) :: fr, h, e)} \quad (10)$$

The instruction `pop` removes the topmost stack operand v, provided it is of type W:

$$\frac{\begin{array}{l} instr(ts) = \texttt{pop} \\ s = (v : W) :: sr \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', sr, l, m) :: fr, h, e)} \quad (11)$$

Similarly, the instruction `pop2` removes the topmost stack operand v, provided it is of type DW. The semantics of the `pop2` instruction is similar to that of the `pop` instruction above (rule 11).

The `pop2` instruction can also be used if the two topmost stack operands v and v' are both of type W:

$$\frac{\begin{array}{l} instr(ts) = \texttt{pop2} \\ s = vs@sr \\ vs = <v, v'> : W^* \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', sr, l, m) :: fr, h, e)} \quad (12)$$

The instruction `swap` swaps the two topmost stack operands v and v', provided both of these are of type W:

$$\frac{\begin{array}{l} instr(ts) = \texttt{swap} \\ s = vs@sr \\ vs = <v, v'> : W^* \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', v' :: v :: sr, l, m) :: fr, h, e)} \quad (13)$$

This can also be viewed as if the `swap` instruction first removes (pops) the topmost stack operand v, and then inserts it below the next (now topmost) operand v' in the stack.

2.5.5 Local Variable Access

The Java Virtual Machine instruction set includes a number of instructions for accessing local variables of different types in the current frame, e.g.

$$\frac{\begin{array}{l} \mathit{instr}(ts) = \texttt{istore}\ j \\ s = (k : \mathit{Int}) :: sr \\ j < \mathit{max}_l(ts) \\ \mathit{succ}(ts) = pc' \end{array}}{ts \Rightarrow ((pc', sr, \mathit{rmDW}(l, j) + \{j \mapsto k\}, m) :: fr, h, e)} \quad (14)$$

The `istore` instruction takes an immediate operand $j \in \mathcal{N}_0$. The topmost stack operand k must be of type Int, and j must be a valid index into the local variables l of the current frame.

The operand k is popped (removed) from the operand stack, k is bound to the local variable at index j, and execution continues with the next instruction in the current method (under the standard assumptions).

Note that the semantic utility function rmDW (see p. 11) is used in the conclusion of the above transition rule for ensuring that, for $j > 1$, the local variable at index $j - 1$ will no longer contain the 'first half' of a two-word value $k' \in \mathit{DW}$.

The semantics of the instructions `istore_<n>` is very similar to that of the `istore` instruction above (rule 14), except that these instructions have an implicit immediate operand of 0, 1, 2, or 3; i.e., the topmost stack operand k is stored into the local variable at index 0, 1, 2, or 3, respectively.

Similarly, the following instructions store the topmost stack operand into a local variable; the semantics of these functions is also very similar to that of the `istore` instruction above (rule 14):

- `fstore` j stores a value of type Float into local variable j
- `fstore_<n>` stores a value of type Float into local variable 0, 1, 2, or 3
- `astore` j stores a value of type Ref_0 or PC into local variable j
- `astore_<n>` stores a value of type Ref_0 or PC into local variable 0, 1, 2, or 3

The instruction `iload` loads the value of a local variable:

$$\frac{\begin{array}{l} \mathit{instr}(ts) = \texttt{iload}\ j \\ l(j) = k : \mathit{Int} \\ \mathit{size}(s) + \mathit{size}(k) \leq \mathit{max}_s(ts) \\ \mathit{succ}(ts) = pc' \end{array}}{ts \Rightarrow ((pc', k :: s, l, m) :: fr, h, e)} \quad (15)$$

The `iload` instruction takes an immediate operand $j \in \mathcal{N}_0$, which must be in the domain of the set of local variables l. The value k of the local variable at index j must be of type Int. This value is pushed onto the operand stack, and execution continues with the next instruction in the current method (under the standard assumptions).

Note that a local variable must generally have been stored into before a value can be loaded from it. However, the parameters of a method are placed in the first local variables of the corresponding frame when the method is invoked (cf. Section 2.5.13).

The semantics of the `iload_<n>` instructions is very similar to that of the `iload` instruction above (rule 15), except that these instructions have an implicit immediate operand of 0, 1, 2, or 3; that is, the value of the local variable at index 0, 1, 2, or 3 is loaded, and pushed onto the operand stack.

Similarly, the following instructions load the value of a local variable, and push that value onto the operand stack; the semantics of these functions is also very similar to that of the `iload` instruction above (rule 15):

- `fload` j loads the value of local variable j, provided it is of type *Float*
- `fload_<n>` loads the value of local variable 0, 1, 2, or 3, provided it is of type *Float*
- `aload` j loads the value of local variable j, provided it is of type Ref_0
- `aload_<n>` loads the value of local variable 0, 1, 2, or 3, provided it is of type Ref_0

Note that the `aload` and `aload_<n>` instructions cannot be used for loading a value of type PC from a local variable, although a value of type PC can be stored into a local variable using the `astore` and `astore_<n>` instructions[3]. A value of type PC stored in a local variable can only be retrieved using the `ret` instruction (cf. rule 39).

The instruction `lstore` can be used for storing a value of type *Long* into a local variable:

$$\frac{\begin{array}{l} instr(ts) = \texttt{lstore}\ j \\ s = (k : Long) :: sr \\ j + 1 < max_l(ts) \\ succ(ts) = pc' \\ rmDW(l, j) = l' \end{array}}{ts \Rightarrow ((pc', sr, l'_{j+1} + \{j \mapsto k\}, m) :: fr, h, e)} \quad (16)$$

The semantics of the `lstore` instruction is similar to that of the `istore` instruction above (rule 14). Note, however, that $j + 1$ must also be a valid index into the local variables of the current frame.

Furthermore, if the local variable at index $j + 1$ contains a value, that local variable entry is removed. This is because the value of type *Long* being stored into the local variable at index j is considered to occupy two local variable entries.

Similarly, the following functions store the value of the topmost stack operand into a local variable; the semantics of these functions is very similar to that of the `lstore` instruction above (rule 16):

- `lstore_<n>` stores a value of type *Long* into local variable 0, 1, 2, or 3
- `dstore` j stores a value of type *Double* into local variable j
- `dstore_<n>` stores a value of type *Double* into local variable 0, 1, 2, or 3

[3] According to *The Java Virtual Machine Specification*[2, p. 160 ff], this asymmetry is intentional.

The instruction `lload` loads a value of type *Long* from a local variable, and pushes it onto the operand stack:

$$\frac{\begin{array}{l} instr(ts) = \texttt{lload}\ j \\ l(j) = k : Long \\ j+1 \notin dom(l) \\ size(s) + size(k) \leq max_s(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', k :: s, l, m) :: fr, h, e)} \quad (17)$$

The semantics of the `lload` instruction is similar to that of the `iload` instruction above (rule 15). Note, however, that the local variable entry at index $j+1$ must be unused. Assuming that any *Long* (or *Double*) value in a local variable will always be removed when a value is stored into the next local variable, the premise $j + 1 \notin dom(l)$ is not strictly necessary.

The following instructions also load the value of a local variable, and push that value onto the operand stack; the semantics of these functions is very similar to that of the `lload` instruction above (rule 17):

- `lload_<n>` loads the value of local variable 0, 1, 2, or 3, provided it is of type *Long*
- `dload` j loads the value of local variable j, provided it is of type *Double*
- `dload_<n>` loads the value of local variable 0, 1, 2, or 3, provided it is of type *Double*

The instruction `iinc` increments the value of a local variable:

$$\frac{\begin{array}{l} instr(ts) = \texttt{iinc}\ j\ k \\ l(j) = k' : Int \\ succ(ts) = pc' \\ iadd(k', k) = k'' \end{array}}{ts \Rightarrow ((pc', s, l + \{j \mapsto k''\}, m) :: fr, h, e)} \quad (18)$$

The `iinc` instruction takes two immediate operands, $j \in \mathcal{N}_0$, and $k \in Int$. The value k' of the local variable at index j must be of type *Int*.

The new value of the local variable at index j is calculated using the abstract semantic function *iadd* (see p. 8). It is assumed that overflow in connection with this calculation must be ignored (cf. Section 3.3.9).

The resulting value k'' is bound to the local variable, and execution proceeds with the next instruction in the current method (under the standard assumptions).

2.5.6 Array Access

At the level of the Java Virtual Machine, instances of array classes are not treated as first-class objects. Array classes are not assumed to declare any members, and the instruction set of the Java Virtual Machine includes a special set of instructions specifically for accessing array objects, e.g.

$$\frac{\begin{array}{l} instr(ts) = \texttt{arraylength} \\ s = (r : Ref) :: sr \\ h(r) = (d, k, av) : Obj_a \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', k :: sr, l, m) :: fr, h, e)} \quad (19)$$

The `arraylength` instruction calculates the lenght of an array object; this corresponds to resolving a reference to the `length` field of an array object in Java.

The topmost stack operand must be a reference to an array object in the heap. The reference r is popped from the operand stack, the length k of the array object is pushed onto the stack, and execution continues with the next instruction in the current method (under the standard assumptions). Note that $size(k) \leq size(r)$ is assumed to hold.

The `bastore` instruction is used for storing a value into a `byte` array:

$$\frac{\begin{array}{l} instr(ts) = \texttt{bastore} \\ s = (k : Int) :: (k' : Int) :: (r : Ref) :: sr \\ h(r) = (d, k'', av) : Obj_a \\ d = (1, \texttt{byte}) : Array \\ (0 : Int) \leq k' < k'' \\ succ(ts) = pc' \\ (d, k'', av + \{k' \mapsto i2b(k)\}) = o : Obj_a \end{array}}{ts \Rightarrow ((pc', sr, l, m) :: fr, h + \{r \mapsto o\}, e)} \quad (20)$$

The two topmost stack operands k and k' must be of type Int, the third-topmost stack operand r must be a reference to a one-dimensional array object of component type `byte`, and the value k' must be a valid index into that array. Note that values of type Int are considered ordered with respect to the $<$ operator.

The three topmost stack operands are popped, the truncated value $i2b(k)$ is bound to the component at index k' of the array object, and execution proceeds with the next instruction of the current method (under the standard assumptions).

Note that the abstract semantic function $i2b$ is used for truncation of the value k being stored into the array, and that the value returned by this function is of type Int. This corresponds to the combined truncation and expansion of an Int value being stored into, and loaded from, a `byte` array (cf. rule 22).

The `bastore` instruction can also be used for storing a value of type Int into a one-dimensional array object of component type `boolean`; when used in this way, the semantics of the `bastore` instruction is very similar to that specified above (rule 20), except that the component type of the array object must be `boolean`, and that the abstract function $i2z$ is used for truncation of the value k being stored.

The semantics of the instructions `castore` and `sastore` is also very similar to that of the `bastore` instruction above (rule 20). However, for `castore`, the array object must be of component type `char`, and the abstract function $i2c$ is used for truncation of the value k being stored; for `sastore`, the array component type must be `short`, and the abstract function $i2s$ is used for truncation of the value being stored.

The instruction `lastore` is used for storing a value into a `long` array:

$$\frac{\begin{array}{l} instr(ts) = \texttt{lastore} \\ s = (k : Long) :: (k' : Int) :: (r : Ref) :: sr \\ h(r) = (d, k'', av) : Obj_a \\ d = (1, \texttt{long}) : Array \\ (0 : Int) \leq k' < k'' \\ succ(ts) = pc' \\ (d, k'', av + \{k' \mapsto k\}) = o : Obj_a \end{array}}{ts \Rightarrow ((pc', sr, l, m) :: fr, h + \{r \mapsto o\}, e)} \quad (21)$$

The semantics of the `lastore` instruction is similar to that of the `bastore` instruction above (rule 20), except that the topmost stack operand k must be of type *Long*, the component type of the array object must be `long`, and the value k is stored unchanged into the array component at index k'.

The following instructions also store the topmost stack operand into a one-dimensional array object of a *Simple* component type; the semantics of these instructions is very similar to that of the `lastore` instruction above (rule 21):

- `iastore` stores a value of type *Int* into an array of component type `int`
- `fastore` stores a value of type *Float* into an array of component type `float`
- `dastore` stores a value of type *Double* into an array of component type `double`

The instruction `baload` can be used for loading a value from a `byte` array:

$$\frac{\begin{array}{l} instr(ts) = \texttt{baload} \\ s = (k : Int) :: (r : Ref) :: sr \\ h(r) = ((1, \texttt{byte}), k', av) : Obj_a \\ av(k) = k'' : Int \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', k'' :: sr, l, m) :: fr, h, e)} \quad (22)$$

The topmost stack operand k must be of type *Int*, the second-topmost stack operand must be a reference r to a one-dimensional array object of component type `byte`, and the value k must be a valid index into that array.

The two topmost stack operands are popped, the value k'' of the array component at index k is pushed onto the stack, and execution continues with the next instruction in the current method (under the standard assumptions).

Note that the value k'' being loaded from a `byte` array is considered to be of type *Int*. In a concrete implementation of the Java Virtual Machine, a component value in an array of a 'small' component type (`boolean`, `char`, `byte`, and `short`) may be represented in a more compact format than that required for a value of type *Int*; when such a component is loaded, it must be extended to a value of type *Int*, before being pushed onto the operand stack.

In this abstract specification of the Java Virtual Machine semantics, the combined operation of truncating a value of type *Int* to a 'smaller' type, and then extending that value to type *Int*, is represented by the abstract semantic functions *i2z*, *i2b*, *i2c*, and *i2s* (see p. 8); these functions are used in connection with instructions storing values of type *Int* into arrays of component type `boolean`, `byte`, `char`, and `short` (cf. rule 20).

Note also, that it is not necessary to store a value into an array component before loading from it; the value being loaded from an array component will be the initial (default) value, if no value has been stored into the component (using an array store instruction).

The `baload` instruction can also be used for loading a value from a `boolean` array; when used in this way, the semantics of the `baload` instruction is similar to that specified above (rule 22), except that the component type of the array object must be `boolean`.

The following instructions also load a value from a one-dimensional array of a *Simple* component type, and push that value onto the operand stack; the semantics of these functions is very similar to that of the `baload` instruction above (rule 22):

- `caload` loads a value of type *Int* from an array of component type `char`

- `saload` loads a value of type *Int* from an array of component type `short`
- `iaload` loads a value of type *Int* from an array of component type `int`
- `laload` loads a value of type *Long* from an array of component type `long`
- `faload` loads a value of type *Float* from an array of component type `float`
- `daload` loads a value of type *Double* from an array of component type `double`

The instruction `aastore` stores a value into an array of a non-*Simple* component type:

$$\frac{\begin{array}{l} instr(ts) = \texttt{aastore} \\ s = (r : Ref_0) :: (k : Int) :: (r' : Ref) :: sr \\ h(r') = (d, k', av) : Obj_a \\ d = (n, t) : Array \\ n = 1 \Rightarrow t \in Id_c \\ compatVal(compDesc(d), r, h, e) \\ (0 : Int) \leq k < k' \\ succ(ts) = pc' \\ (d, k', av + \{k \mapsto r\}) = o : Obj_a \end{array}}{ts \Rightarrow ((pc', sr, l, m) :: fr, h + \{r' \mapsto o\}, e)} \quad (23)$$

The semantics of the `aastore` instruction is similar to that of the other array store instructions, e.g. `bastore` (cf. rule 20). However, the topmost stack operand r must be assignment compatible with the component type of the array object being referenced by the third-topmost stack operand r', according to the predicate *compatVal* (see p. 18). The component type of that array object must not be a *Simple* type; that is, if the array is one-dimensional, the base type must be a class or interface type.

The instruction `aaload` loads a value from an array of a non-*Simple* component type:

$$\frac{\begin{array}{l} instr(ts) = \texttt{aaload} \\ s = (k : Int) :: (r : Ref) :: sr \\ h(r) = ((n, t), k', av) : Obj_a \\ n = 1 \Rightarrow t \in Id_c \\ av(k) = r' : Ref_0 \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', r' :: sr, l, m) :: fr, h, e)} \quad (24)$$

The semantics of the `aaload` instruction is similar to that of the other array load instructions, e.g. `baload` (cf. rule 22). The component type of the array object referenced by the second-topmost stack operand r must, however, not be a *Simple* type.

Note that $size(r') \leq size(k) + size(r)$ is assumed to hold.

2.5.7 Arithmetic Operations

The instruction set of the Java Virtual Machine contains a few instructions for performing arithmetic operations on stack operands of different types. It is assumed that these instructions never cause an exception to be thrown in connection with the calculation of their results. Note, however, that the instructions `idiv`, `irem`, `ldiv`, and `lrem` are supposed to throw an exception in case of division by zero, cf. rule 26 ff.

The instruction `iadd` performs integer addition of two operands:

$$\frac{\begin{array}{l} \mathit{instr}(ts) = \texttt{iadd} \\ s = (k : \mathit{Int}) :: (k' : \mathit{Int}) :: sr \\ \mathit{succ}(ts) = pc' \end{array}}{ts \Rightarrow ((pc', \mathit{iadd}(k, k') :: sr, l, m) :: fr, h, e)} \quad (25)$$

The two topmost stack operands k and k' must be of type *Int*; these are popped from the operand stack, the value $\mathit{iadd}(k, k')$ is pushed onto the stack, and execution continues with the next instruction in the current method (under the standard assumptions).

Note that the sum $k + k'$ is calculated using the abstract semantic function *iadd* (see p. 8); overflow in connection with this calculation is ignored.

Similarly, each of the following instructions perform an arithmetic operation on the two topmost stack operands k and k', using abstract semantic functions for calculating the result; the semantics of these instructions is very similar to that of the `iadd` instruction above (rule 25):

- `imul`, `isub`, `iand`, `ior`, `ixor`, `ishl`, `ishr`, and `iushr` each operate on two operands of type *Int*, using the functions *imul*, *isub*, *iand*, *ior*, *ixor*, *ishl*, *ishr*, and *iushr*, respectively
- `ladd`, `lmul`, `lsub`, `land`, `lor`, and `lxor` each operate on two operands of type *Long*, using the functions *ladd*, *lmul*, *lsub*, *land*, *lor*, and *lxor*, respectively
- `fadd`, `fdiv`, `fmul`, `frem`, and `fsub` each operate on two operands of type *Float*, using the functions *fadd*, *fdiv*, *fmul*, *frem*, and *fsub*, respectively
- `dadd`, `ddiv`, `dmul`, `drem`, and `dsub` each operate on two operands of type *Double*, using the functions *dadd*, *ddiv*, *dmul*, *drem*, and *dsub*, respectively

The above `add`, `div`, `mul`, `rem` and `sub` instruction variants perform addition, division (k'/k), multiplication, remainder calculation (with respect to k'/k), and subtraction ($k' - k$), respectively. Note that the abstract functions *fdiv*, *frem*, *ddiv*, and *drem* are assumed to return the special value `NaN` (of the proper type) in case of division by zero.

The `and`, `or` and `xor` instruction variants perform bitwise logical conjunction, inclusive disjunction, and exclusive disjunction, respectively.

The `ishl` instruction shifts the value k' left by k bit positions, `ishr` shifts k' right by k bit positions (with sign-extension), and `iushr` shifts k' right by k bit positions (with zero-extension).

The semantics of the instructions `lshl`, `lshr`, and `lushr` (not mentioned above) is very similar to that of the instructions `ishl`, `ishr`, and `iushr`, except that the second-topmost stack operand k' must be of type *Long*, and that the abstract functions *lshl*, *lshr*, and *lushr*, respectively, are used for calculating the resulting value.

The instruction `idiv` performs integer division of two operands of type *Int*:

$$\frac{\begin{array}{l} \mathit{instr}(ts) = \texttt{idiv} \\ s = (k : \mathit{Int}) :: (k' : \mathit{Int}) :: sr \\ k \neq (0 : \mathit{Int}) \\ \mathit{succ}(ts) = pc' \end{array}}{ts \Rightarrow ((pc', \mathit{idiv}(k', k) :: sr, l, m) :: fr, h, e)} \quad (26)$$

The semantics of the `idiv` instruction is similar to that of the `iadd` instruction above (rule 25), except that the topmost stack operand k must not be zero, and that the abstract function *idiv* is used for calculating the quotient k'/k rounded to an integer of type *Int*.

The semantics of the instruction `irem` is very similar to that of the `idiv` instruction above (rule 26), except that it uses the abstract function *irem* for calculating the remainder with respect to k'/k.

The semantics of the instructions `ldiv` and `lrem` is also very similar to that of the `idiv` instruction above (rule 26). However, the two topmost stack operands must be of type *Long*; for `ldiv`, the abstract function *ldiv* is used for calculating the quotient k'/k rounded to an integer of type *Long*; and for `lrem`, the abstract function *lrem* is used for calculating the remainder with respect to k'/k.

Note that the above specification of the semantics of the instructions `idiv`, `irem`, `ldiv`, and `lrem` does not cover division by zero. However, *The Java Virtual Machine Specification*[2] governs that those instructions throw an instance of class `java.lang.ArithmeticException` in case of division by zero. In this abstract specification of the Java Virtual Machine semantics, division by zero is considered an error; see section 2.6 for a discussion of error semantics.

The instruction `ineg` negates the topmost stack operand, provided it is of type *Int*:

$$\frac{\begin{array}{l} \mathit{instr}(ts) = \texttt{ineg} \\ s = (k : \mathit{Int}) :: sr \\ \mathit{succ}(ts) = pc' \end{array}}{ts \Rightarrow ((pc', \mathit{ineg}(k) :: sr, l, m) :: fr, h, e)} \quad (27)$$

The semantics of the instructions `lneg`, `fneg`, and `dneg` is very similar to that of the `ineg` instruction above (rule 27), except that the topmost stack operand must be of type *Long*, *Float*, or *Double*, respectively, and that the abstract functions *lneg*, *fneg*, and *dneg*, respectively, are used for calculating the negated value.

2.5.8 Conversion

The Java Virtual Machine instruction set features a number of instructions for conversion between and truncation of operands of simple types, e.g.

$$\frac{\begin{array}{l} \mathit{instr}(ts) = \texttt{i2b} \\ s = (k : \mathit{Int}) :: sr \\ \mathit{i2b}(k) = k' : \mathit{Int} \\ \mathit{succ}(ts) = pc' \end{array}}{ts \Rightarrow ((pc', k' :: sr, l, m) :: fr, h, e)} \quad (28)$$

The instruction `i2b` truncates a value of type *Int* to a value suitable for a Java variable of type `byte`. The topmost stack operand k must be of type *Int*; this value is popped from the stack, the truncated value k' is pushed onto the stack, and execution proceeds with the next instruction in the current method (under the standard assumptions).

Note that the abstract method *i2b* (see p. 8) is used for calculating the truncated value k'.

The following instructions also truncate the topmost stack operand, or convert it to a different simple type; the semantics of these instructions is very similar to that of the `i2b` instruction above (rule 28):

- `i2c` and `i2s` truncate a value of type *Int* to a value suitable for a Java variable of type `char` or `short`, respectively; the abstract functions *i2c* and *i2s*, respectively, are used for calculation of the truncated values
- `i2f` converts a value of type *Int* to type *Float*, using the abstract function *i2f*
- `l2i` and `l2f` truncate a value of type *Long* to type *Int* or *Float*, respectively, using the abstract functions *l2i* and *l2f*, respectively
- `l2d` converts a value of type *Long* to type *Double*, using the abstract function *l2d*
- `f2i` truncates a value of type *Float* to type *Int*, using the abstract function *f2i*
- `d2i`, `d2l`, and `d2f` truncate a value of type *Double* to type *Int*, *Long*, or *Float*, respectively, using the abstract functions *d2i*, *d2l*, and *d2f*, respectively

Note that $size(k') \leq size(k)$ is assumed to hold for all of the above conversion instructions.

The instruction `i2l` converts an operand of type *Int* to type *Long*:

$$\frac{\begin{array}{l} instr(ts) = \texttt{i2l} \\ s = (k : Int) :: sr \\ i2l(k) = k' : Long \\ size(sr) + size(k') \leq max_s(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', k' :: sr, l, m) :: fr, h, e)} \quad (29)$$

The semantics of the `i2l` instruction is similar to that of the `i2b` instruction above, except for the proviso that the operand stack does not overflow ($size(k') > size(k)$ is assumed to hold).

The semantics of the following instructions is very similar to that of the `i2l` instruction above (rule 29):

- `i2d` converts a value of type *Int* to type *Double*, using the abstract function *i2d*
- `f2l` truncates a value of type *Float* to type *Long*, using the abstract function *f2l*
- `f2d` converts a value of type *Float* to type *Double*, using the abstract function *f2d*

2.5.9 Comparison

The instruction `lcmp` compares two operands of type *Long*:

$$\frac{\begin{array}{l} instr(ts) = \texttt{lcmp} \\ s = (k : Long) :: (k' : Long) :: sr \\ lcmp(k', k) = k'' : Int \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', k'' :: sr, l, m) :: fr, h, e)} \quad (30)$$

The two topmost stack operands k and k' must be of type *Long*. The result k'' of comparing k' to k is calculated using the abstract function *lcmp* (see p. 8). The two topmost stack operands are popped, the result k'' of the comparison is pushed, and execution continues with the next instruction in the current method (under the standard assumptions).

The instructions `fcmpg` and `fcmpl` each compare two operands of type *Float*:

$$\frac{\begin{array}{l} \mathit{instr}(ts) \in \{\texttt{fcmpg}, \texttt{fcmpl}\} \\ s = (k : \mathit{Float}) :: (k' : \mathit{Float}) :: sr \\ k \neq \texttt{NaN} \ \wedge \ k' \neq \texttt{NaN} \\ \mathit{fcmp}(k', k) = k'' : \mathit{Int} \\ \mathit{succ}(ts) = pc' \end{array}}{ts \Rightarrow ((pc', k'' :: sr, l, m) :: fr, h, e)} \quad (31)$$

The semantics of the `fcmpg` and `fcmpl` instructions is similar to that of the `lcmp` instruction above (rule 30). However, the abstract function *fcmp* is used for comparing k' to k, and the special value `NaN` must be treated specially:

$$\frac{\begin{array}{l} \mathit{instr}(ts) = \texttt{fcmpg} \\ s = (k : \mathit{Float}) :: (k' : \mathit{Float}) :: sr \\ k = \texttt{NaN} \ \vee \ k' = \texttt{NaN} \\ (1 : \mathit{Int}) = k'' \\ \mathit{succ}(ts) = pc' \end{array}}{ts \Rightarrow ((pc', k'' :: sr, l, m) :: fr, h, e)} \quad (32)$$

If k or k'' is `NaN`, `fcmpg` pushes the value $k'' = 1$ of type *Int*, whereas `fcmpl` pushes the *Int* value $k'' = -1$. Note that the semantics of the `fcmpl` instruction only differs from that of the `fcmpg` instruction with respect to the treatment of `NaN` operands.

The semantics of the instructions `dcmpg` and `dcmpl` is very similar to that of the `fcmpg` and `fcmpl` instructions above, except that the two topmost stack operands must be of type *Double*, and that the abstract function *dcmp* is used for comparison of non-`NaN` values.

Note that $\mathit{size}(k'') \leq \mathit{size}(k) + \mathit{size}(k')$ is assumed to hold for all of the above comparison instructions.

2.5.10 Branching

The instruction set of the Java Virtual Machine includes a number of instructions for branching to an instruction in the current method, e.g.

$$\frac{\begin{array}{l} \mathit{instr}(ts) = \texttt{goto}\ \delta \\ \mathit{jump}(ts, \delta) = pc' \end{array}}{ts \Rightarrow ((pc', s, l, m) :: fr, h, e)} \quad (33)$$

The instruction `goto` takes an immediate operand $\delta \in \mathit{Offset}$. Provided the instruction address pc' at offset δ from the current program counter pc is in the set of valid target addresses for the thread state ts, execution continues with the instruction at address pc'.

The instruction `ifeq` conditionally branches to an instruction at a specified offset from the current program counter pc:

$$\frac{\begin{array}{l} \mathit{instr}(ts) = \texttt{ifeq}\ \delta \\ s = (k : \mathit{Int}) :: sr \\ \mathit{jump}(ts, \delta) = pc' \\ \mathit{succ}(ts) = pc'' \\ \mathit{if}(k = (0 : \mathit{Int}), pc', pc'') = pc''' \end{array}}{ts \Rightarrow ((pc''', sr, l, m) :: fr, h, e)} \quad (34)$$

The semantics of the `ifeq` instruction differs from that of the `goto` instruction above (rule 33), in that the address pc'' of the next instruction in the current method must also be in the set of valid target addresses, and that the topmost stack operand k must be of type Int. The value k is popped off the operand stack; if k is zero, execution proceeds with the instruction at pc'; otherwise, execution continues with the next instruction (at pc'').

The following instructions also conditionally branch to the instruction specified by the immediate operand δ, depending on the value of the topmost stack operand k; the semantics of these instructions is very similar to that of the `ifeq` instruction above (rule 34):

- `ifne`, `iflt`, `ifle`, `ifgt`, and `ifge` branch to the instruction at offset δ from the current program counter if $k \neq 0$, if $k < 0$, if $k \leq 0$, if $k > 0$, and if $k \geq 0$, respectively; k must be of type Int
- `ifnull` branches to the instruction at offset δ if k, which must be of type Ref_0, equals the special value `null`
- `ifnonnull` branches to the instruction at offset δ if k, which must be of type Ref_0, is not the special value `null`

The instruction `if_icmpeq` compares two operands, and conditionally branches to an instruction at a specified offset from the current program counter:

$$\frac{\begin{array}{l} instr(ts) = \texttt{if\_icmpeq}\ \delta \\ s = (k : Int) :: (k' : Int) :: sr \\ jump(ts, \delta) = pc' \\ succ(ts) = pc'' \\ if(k' = k, pc', pc'') = pc''' \end{array}}{ts \Rightarrow ((pc''', sr, l, m) :: fr, h, e)} \quad (35)$$

The semantics of the `if_icmpeq` instruction differs from that of the `ifeq` instruction above (rule 34), in that the two topmost stack operands k and k' must both be of type Int. If $k' = k$, execution continues with the instruction at pc'; otherwise, execution proceeds with the next instruction.

The following instructions also conditionally branch to the instruction specified by the immediate operand δ, depending on the value of the two topmost stack operands k and k'; the semantics of these instructions is very similar to that of the `if_icmpeq` instruction above (rule 35):

- `if_icmpne`, `if_icmplt`, `if_icmple`, `if_icmpgt`, `if_icmpge` branch to the instruction at offset δ if $k' \neq k$, if $k' < k$, if $k' \leq k$, if $k' > k$, and if $k' \geq k$, respectively; k and k' must be of type Int
- `if_acmpeq` branches to the instruction at offset δ if k' and k, which must be of type Ref_0, are equal
- `if_acmpne` branches to the instruction at offset δ if k' and k, which must be of type Ref_0, are not equal

Note that the references to two objects are considered equal if, and only if, the references are themselves equal.

The instruction `lookupswitch` selects one of a number of offset values, and branches to the instruction at that offset from the current program counter:

$$\frac{\begin{array}{l}instr(ts) = \texttt{lookupswitch}\ \delta\ \theta \\ s = (k : Int) :: sr \\ switch(k, \theta, \delta) = \delta' \\ jump(ts, \delta') = pc'\end{array}}{ts \Rightarrow ((pc', sr, l, m) :: fr, h, e)} \quad (36)$$

The `lookupswitch` instruction takes as immediate operands a (default) offset $\delta \in Offset$, and a 'jump table' $\theta \in Int \to Offset$.

The topmost stack operand k must be of type Int; the semantic function *switch* (see p. 11) is used for selecting the offset δ' corresponding to the key k (if k is not in the domain of θ, the default offset δ is used).

The value k is popped from the operand stack, and execution continues with the instruction at the selected offset δ' from the current program counter, provided that the target address (pc') is in the set of valid target addresses for the thread state ts.

At the abstract level of this specification, the semantics of the instruction `tableswitch` is similar to that of the `lookupswitch` instruction above (rule 36):

$$\frac{\begin{array}{l}instr(ts) = \texttt{tableswitch}\ \delta\ k\ \theta \\ s = (k' : Int) :: sr \\ switch(k' - k, \theta, \delta) = \delta' \\ jump(ts, \delta') = pc'\end{array}}{ts \Rightarrow ((pc', sr, l, m) :: fr, h, e)} \quad (37)$$

However, the `tableswitch` instruction takes an extra immediate operand $k \in Int$; this operand is used as a 'displacement' for the jump table θ.

In the concrete representation of the `tableswitch` instruction, the immediate operand θ is represented as a (zero-based) array of offset values; the offset corresponding to a specific key k' can thus be found at index $k' - k$ in the array (using the default offset δ if $k' - k$ is not a valid index into the array).

In the concrete representation of the `lookupswitch` instruction, on the other hand, the immediate operand θ is represented a (sorted) association table, containing a number of (key, offset) pairs; the offset corresponding to a specific key can thus be found by searching through the association table (using the default offset δ if the key is not found).

Note that the calculation of a jump offset during evaluation of the `tableswitch` instruction above (rule 37) involves the abstract subtraction $k' - k$. It is assumed that this calculation must not overflow (cf. Section 3.3.9); hence, the naïve approach of using $isub(k', k)$ would not suffice. In a concrete implementation of the Java Virtual Machine, overflow may be avoided in a number of ways, e.g. by converting k' and k to values of type $Long$, and then use $lsub$ for the subtraction.

The instruction `jsr` branches to a 'subroutine' within the code array of the current method:

$$\frac{\begin{array}{l}instr(ts) = \texttt{jsr}\ \delta \\ size(s) + size(pc) \leq max_s(ts) \\ jump(ts, \delta) = pc'\end{array}}{ts \Rightarrow ((pc', pc :: s, l, m) :: fr, h, e)} \quad (38)$$

The semantics of the `jsr` instruction is similar to that of the `goto` instruction above (rule 33), except that the current program counter is pushed onto the operand stack, and that it is verified that the stack does not thereby overflow.

The instruction `ret` returns from a subroutine:

$$\frac{\begin{array}{l} instr(ts) = \texttt{ret}\ j \\ l(j) = pc' : PC \\ succ(((pc', s, l, m) :: fr, h, e) : TS) = pc'' \end{array}}{ts \Rightarrow ((pc'', s, l, m) :: fr, h, e)} \quad (39)$$

The `ret` instruction takes an immediate operand $j \in \mathcal{N}_0$, which must be in the domain of the set of local variables l; the value pc' of the local variable at index j must be of type PC. Execution continues with the instruction at the next address after pc', provided it is in the set of valid target addresses.

The proviso $succ(\ldots) = pc''$ in the premises for the `ret` instruction above (rule 39) corresponds to a dynamic verification (at run-time) that

> *Execution never falls off the bottom of the code array.*
> (*The Java Virtual Machine Specification*[2, p. 123])

If a similar proviso had instead occurred in the premises of the `jsr` instruction (rule 38), this would have corresponded to a static (load-time) verification of the same property.

2.5.11 Class and Array Instantiation

The instruction `new` instantiates a specified class:

$$\frac{\begin{array}{l} instr(ts) = \texttt{new}\ i \\ pool(ts)(i) = id_c' : Id_c \\ e(id_c') = ((acc_c, id_c'', is, fd, cv, md, mi, cp), sv) \\ acc_c \cap \{\texttt{interface}, \texttt{abstract}\} = \emptyset \\ access(id_c', acc_c, id_c) \\ fields(id_c', e) = iv \\ (id_c', iv) = o : Obj_u \\ r \in Ref \backslash dom(h) \\ size(s) + size(r) \leq max_s(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', r :: s, l, m) :: fr, h + \{r \mapsto o\}, e)} \quad (40)$$

The `new` instruction takes an immediate operand $i \in Index$. If it holds that

- i is a valid index into the constant pool of the current class
- the constant pool entry at index i is a class or interface reference id_c'
- the declaration of a class named id_c' has been loaded into the environment e
- class id_c' is not flagged as being `abstract`
- the current class is granted access to class id_c', according to the *access* predicate (see p. 19)

then the instance fields of class $id_c{}'$ and all its superclasses are initialized using the utility function *fields* (see p. 13); an instance of class $id_c{}'$, with instance field values iv, is allocated in the heap (using a fresh heap entry); a reference to the new, uninitialized object o is pushed onto the operand stack, and execution proceeds with the next instruction in the current method (under the standard assumptions).

Note that a newly created class instance is tagged as being uninitialized; this restricts the access to members of the object, until the object has been initialized by invocation of a constructor (cf. rule 53).

The instruction `newarray` creates a one-dimensional array of a *Simple* component type:

$$\frac{\begin{array}{l} instr(ts) = \texttt{newarray}\ t \\ s = (k : Int) :: sr \\ k \geq (0 : Int) \\ singleArray(arrayDesc(t), k, h) = (r, h') \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', r :: sr, l, m) :: fr, h', e)} \quad (41)$$

The `newarray` instruction takes an immediate operand $t \in Simple$; the topmost stack operand k must be of type Int, and must not be negative.

The utility function *singleArray* (see p. 14) is used for allocating a fresh array object of component type t and length k in the heap; this yields a reference r to the new object, and an extended heap h'. The reference r is pushed onto the operand stack, and execution continues with the next instruction in the current method (under the standard assumptions). Note that $size(r) \leq size(k)$ is assumed to hold.

Note also that array objects are considered to be instances of class `java.lang.Object`. Since that class is not assumed to declare any instance fields, and since an array class does not declare any members by itself, an array object does not include any instance field values.

It is assumed that a fresh array object must be created, even if the specified length k of the array to be created is zero (cf. Section 3.3.3).

The instruction `anewarray` creates (one dimension of) an array of a non-*Simple* component type:

$$\frac{\begin{array}{l} instr(ts) = \texttt{anewarray}\ i \\ pool(ts)(i) = d : (Array \cup Id_c) \\ d = (n, t) : Array \Rightarrow n < 255 \\ s = (k : Int) :: sr \\ k \geq (0 : Int) \\ singleArray(arrayDesc(d), k, h) = (r, h') \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', r :: sr, l, m) :: fr, h', e)} \quad (42)$$

The semantics of the `anewarray` instruction differs from that of the `newarray` instruction above (rule 41), in that it takes an immediate operand $i \in Index$, which must be a valid index into the constant pool of the current class; the constant pool entry d at index i must be an array class reference, or a class or interface reference; if d is an array class reference, it must have less than 255 dimensions.

In case the base type of the array being created is a class or interface type, it is assumed that the declaration of that class or interface need not have been loaded into the environment

e, and that the current class need not have permission to access that class or interface (cf. Section 3.3.3).

The instruction `multianewarray` creates one or more dimensions of an array (of *Simple* or non-*Simple* component type):

$$\frac{\begin{array}{l} instr(ts) = \texttt{multianewarray}\ i\ n \\ pool(ts)(i) = d : Array \\ d = (n', t) \\ n \leq n' \leq 255 \\ s = ks@sr \\ ks = <k_1, k_2, k_3, \ldots, k_n> : Int^* \\ \forall 1 \leq j \leq n.k_j \geq (0 : Int) \\ newArray(d, ks, h) = (r, h') \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', r :: sr, l, m) :: fr, h', e)} \quad (43)$$

The `multianewarray` instruction takes two immediate operands, $i \in Index$, and $n \in \mathcal{N}$. If it holds that

- i is a valid index into the constant pool of the current class
- the constant pool entry at index i is an array class reference d
- the number of dimensions n' of that array class is greater than or equal to the number of dimensions to be created (n), and no greater than 255
- the n topmost stack operands ks are of type Int and non-negative

then the utility function *newArray* (see p. 14) is used for creating the first n dimensions of the specified array type; this yields a reference r to the array object representing the first dimension of the array, and an extended heap h'. The reference r is pushed onto the operand stack, and execution proceeds with the next instruction in the current method (under the standard assumptions). Note that $size(r) \leq size(ks)$ is assumed to hold.

In case the base type of the array being created is a class or interface type, it is assumed that the declaration of that class or interface need not have been loaded into the environment e, and that the current class need not have permission to access that class or interface (cf. Section 3.3.3).

2.5.12 Field Access

The instruction `putstatic` stores an operand into a class field:

$$\frac{\begin{array}{l} instr(ts) = \texttt{putstatic}\ i \\ pool(ts)(i) = (id_c{}', id_f, d) : Const_f \\ e(id_c{}') = (c, sv) \\ c = (acc_c, id_c{}'', is, fd, cv, md, mi, cp) \\ \texttt{interface} \notin acc_c \\ access(id_c{}', acc_c, id_c) \\ fd(id_f) = (acc_f, d') \\ acc_f \cap \{\texttt{static}, \texttt{final}\} = \{\texttt{static}\} \\ \texttt{private} \in acc_f \Rightarrow id_c = id_c{}' \\ \texttt{protected} \in acc_f \Rightarrow id_c{}' \in supers(id_c, e) \\ d = d' \\ s = (v : V) :: sr \\ initialized(v, h) \\ compatVal(d, v, h, e) \\ succ(ts) = pc' \\ sv + \{id_f \mapsto v\} = sv' \\ e + \{id_c{}' \mapsto (c, sv')\} = e' \end{array}}{ts \Rightarrow ((pc', sr, l, m) :: fr, h, e')} \quad (44)$$

The `putstatic` instruction takes an immediate operand $i \in Index$. If it holds that

- i is a valid index into the constant pool of the current class
- the constant pool entry at index i is a symbolic field reference, referring to a field named id_f, with type descriptor d, in class (or interface) $id_c{}'$
- the declaration of a class named $id_c{}'$ has been loaded into the environment e
- the current class has permission to access class $id_c{}'$
- class $id_c{}'$ declares a field named id_f of the same type as that specified by the descriptor d
- the field id_f is declared to be `static`, but not `final`
- the field id_f is not declared to be `private`, unless the current class is the same as that declaring the field
- the field id_f is not declared to be `protected`, unless the current class is the same as that declaring the field, or a subclass thereof
- the topmost stack operand v is of type V, and is not a reference to an uninitialized class instance in the heap h
- v is assignment compatible with the type descriptor d of the field, according to the *compatVal* predicate (see p. 18)

then the value v is bound to the specified class field, the topmost stack operand is popped, and execution continues with the next instruction in the current method (under the standard assumptions).

Note that the value v is not being trunctated before being stored into the class field; it is assumed that a stack operand of type *Int* may be stored unchanged into a field of type `boolean`, `char`, `byte`, or `short`. It is also assumed that a field may not be stored into if it is declared to be `final` (cf. Section 3.3.5).

The instruction `getstatic` loads the value of a class or interface field:

$$\frac{\begin{array}{l} instr(ts) = \texttt{getstatic}\ i \\ pool(ts)(i) = (id_c{}', id_f, d) : Const_f \\ e(id_c{}') = ((acc_c, id_c{}'', is, fd, cv, md, mi, cp), sv) \\ access(id_c{}', acc_c, id_c) \\ fd(id_f) = (acc_f, d') \\ \texttt{static} \in acc_f \\ \texttt{private} \in acc_f \Rightarrow id_c = id_c{}' \\ \texttt{protected} \in acc_f \Rightarrow id_c{}' \in supers(id_c, e) \\ d = d' \\ sv(id_f) = v : V \\ size(s) + size(v) \leq max_s(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', v :: s, l, m) :: fr, h, e)} \quad (45)$$

The semantics of the `getstatic` instruction differs from that of the `putstatic` instruction above (rule 44), in that the referenced field may be declared by an interface, and may be `final`, and that the contents of the operand stack are ignored. The value of the field id_f is pushed onto the operand stack, and execution proceeds with the next instruction in the current method (under the standard assumptions).

Note that it is not necessary to store a value into a field before loading from it; the value being loaded from a field will be the initial (default) value, if no value has been stored into the field.

Note also that loading the value of an interface field is assumed to be permitted (cf. Section 3.3.5).

The instruction `putfield` stores an operand into an instance field of a class instance:

$$\frac{\begin{array}{l} instr(ts) = \texttt{putfield}\ i \\ pool(ts)(i) = (id_c{}', id_f, d) : Const_f \\ e(id_c{}') = ((acc_c, id_c{}'', is, fd, cv, md, mi, cp), sv) \\ \texttt{interface} \notin acc_c \\ access(id_c{}', acc_c, id_c) \\ fd(id_f) = (acc_f, d') \\ acc_f \cap \{\texttt{static}, \texttt{final}\} = \emptyset \\ \texttt{private} \in acc_f \Rightarrow id_c = id_c{}' \\ d = d' \\ s = (v : V) :: (r : Ref) :: sr \\ initialized(v, h) \\ h(r) = (id_c{}''', iv) : Obj_c \\ id_c{}' \in supers(id_c{}''', e) \\ \texttt{protected} \in acc_f \Rightarrow (id_c{}' \in supers(id_c, e)\ \wedge\ id_c \in supers(id_c{}''', e)) \\ compatVal(d, v, h, e) \\ succ(ts) = pc' \\ (id_c{}''', iv + \{(id_c{}', id_f) \mapsto v\}) = o : Obj_c \end{array}}{ts \Rightarrow ((pc', sr, l, m) :: fr, h + \{r \mapsto o\}, e)} \quad (46)$$

The semantics of the `putfield` instruction differs from that of the `putstatic` instruction above (rule 44), in that the referenced field must not be `static`, and that the second-topmost stack operand r must be a reference to an initialized instance of class $id_c{}'$ in the heap. If the referenced instance field is declared to be `protected`, then the current class must be the same as that declaring the field, or a subclass thereof, and the class of the object referenced by r must be the same as the current class, or a subclass thereof.

It is assumed that the class of an object, and all superclasses of that class, are already loaded when there is a reference to the object on the operand stack.

The instruction `getfield` loads the value of an instance field:

$$\frac{\begin{array}{l} instr(ts) = \texttt{getfield}\ i \\ pool(ts)(i) = (id_c{}', id_f, d) : Const_f \\ e(id_c{}') = ((acc_c, id_c{}'', is, fd, cv, md, mi, cp), sv) \\ \texttt{interface} \notin acc_c \\ access(id_c{}', acc_c, id_c) \\ fd(id_f) = (acc_f, d') \\ \texttt{static} \notin acc_f \\ \texttt{private} \in acc_f \Rightarrow id_c = id_c{}' \\ d = d' \\ s = (r : Ref) :: sr \\ h(r) = (id_c{}''', iv) : Obj_c \\ id_c{}' \in supers(id_c{}''', e) \\ \texttt{protected} \in acc_f \Rightarrow (id_c{}' \in supers(id_c, e)\ \wedge\ id_c \in supers(id_c{}''', e)) \\ iv(id_c{}', id_f) = v : V \\ size(sr) + size(v) \leq max_s(ts) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', v :: sr, l, m) :: fr, h, e)} \quad (47)$$

The semantics of the `getfield` instruction differs from that of the `putfield` instruction above (rule 46), in that the referenced field may be `final`, and that the topmost stack operand r must be a reference to an instance of class $id_c{}'$ in the heap; the rest of the operand stack is ignored.

The topmost stack operand is popped, the value v of the instance field is pushed onto the operand stack, and execution continues with the next instruction in the current method (under the standard assumptions).

2.5.13 Normal Method Invocation

In this section we describe invocation of methods implemented via Java byte code. The method invocation instructions specified in the following transition rules can, however, also be used for invoking methods implemented as native methods, cf. Section 2.5.15.

The instruction `invokestatic` invokes a class method:

$$\frac{\begin{array}{l} instr(ts) = \texttt{invokestatic}\ i \\ pool(ts)(i) = (id_c{}', sig', d) : Const_m \\ e(id_c{}') = ((acc_c, id_c{}'', is, fd, cv, md, mi, cp), sv) \\ \texttt{interface} \notin acc_c \\ access(id_c{}', acc_c, id_c) \\ sig' = (id_m, <d_1, d_2, d_3, \ldots, d_k>) \\ id_m \notin \{\texttt{<init>}, \texttt{<clinit>}\} \\ md(sig') = (acc_m, d', excs) \\ acc_m \cap \{\texttt{static}, \texttt{abstract}, \texttt{native}\} = \{\texttt{static}\} \\ \texttt{private} \in acc_m \Rightarrow id_c = id_c{}' \\ \texttt{protected} \in acc_m \Rightarrow id_c{}' \in supers(id_c, e) \\ d = d' \\ mi(sig') = (n_s, n_l, code', hdls') \\ s = as@sr \\ as = <a_k, a_{k-1}, a_{k-2}, \ldots, a_2, a_1> \\ size(as) \leq n_l \\ \forall 1 \leq j \leq k.(initialized(a_j, h) \wedge compatVal(d_j, a_j, h, e)) \\ (min_{pc}(code'), <>, args(as), (id_c{}', sig')) = f' : Frame \end{array}}{ts \Rightarrow (f' :: (pc, sr, l, m) :: fr, h, e)} \quad (48)$$

The `invokestatic` instruction takes an immediate operand $i \in Index$. If it holds that

- i is a valid index into the constant pool of the current class
- the constant pool entry at index i is a symbolic method reference, refererring to a method in class $id_c{}'$ with signature sig' and return type descriptor d
- the declaration of a class named $id_c{}'$ has been loaded into the environment e
- the current class has permission to access class $id_c{}'$
- the referenced method is not a constructor (named `<init>`) or a static initializer (named `<clinit>`)
- class $id_c{}'$ declares and implements a method with signature sig' and the same return type descriptor as that specified by the symbolic method reference

- the method is declared to be `static`, but not `abstract` or `native`
- the method is not declared to be `private`, unless the current class is the same as that declaring the method
- the method is not declared to be `protected`, unless the current class is the same as that declaring the method, or a subclass thereof
- the k first stack operands as are assignment compatible with the corresponding k parameter type descriptors of the method signature sig'
- the total size of the method arguments as is no greater than the local variable limit n_l of the method

then the method arguments as are popped from the operand stack of the current frame, a new frame f' representing the invoked method is pushed onto the frame stack, and execution continues with the first instruction in the invoked method.

The frame f' initially contains an empty operand stack, and the values of the method parameters in the first local variables (initialized by means of the utility function *args*; see p. 12). Note that $min_{pc}(code')$ is assumed to be a valid initial program counter for the new thread state.

The instruction `invokevirtual` invokes an instance method relative to a specified class instance; this is the usual form of dynamic method dispatch:

$$\frac{\begin{array}{l} instr(ts) = \texttt{invokevirtual}\ i \\ pool(ts)(i) = (id_c', sig', d) : Const_m \\ e(id_c') = ((acc_c, id_c'', is, fd, cv, md, mi, cp), sv) \\ \texttt{interface} \notin acc_c \\ sig' = (id_m, <d_1, d_2, d_3, \ldots, d_k>) \\ id_m \notin \{\texttt{<init>}, \texttt{<clinit>}\} \\ sig' \in dom(md) \\ s = as@sr \\ as = <a_k, a_{k-1}, a_{k-2}, \ldots, a_2, a_1>@<r : Ref> \\ \forall 1 \leq j \leq k.(initialized(a_j, h) \wedge compatVal(d_j, a_j, h, e)) \\ h(r) = (id_c''', iv) : Obj_c \\ id_c' \in supers(id_c''', e) \\ lookup(sig', id_c''', e) = (id_c'''', acc_m, d', n_l, code') \\ access(id_c'''', id_c, e) \\ acc_m \cap \{\texttt{private}, \texttt{static}, \texttt{abstract}, \texttt{native}\} = \emptyset \\ \texttt{protected} \in acc_m \Rightarrow (id_c' \in supers(id_c, e) \ \wedge \ id_c \in supers(id_c''', e)) \\ d = d' \\ size(as) \leq n_l \\ (min_{pc}(code'), <>, args(as), (id_c'''', sig')) = f' : Frame \end{array}}{ts \Rightarrow (f' :: (pc, sr, l, m) :: fr, h, e)} \quad (49)$$

The semantics of the `invokevirtual` instruction is similar to that of the `invokestatic` instruction above (rule 48). However,

- the current class is not required to have permission to access class id_c'

- class $id_c{}'$ must declare, but may not implement, a method with signature sig'
- the method arguments $a_k, \ldots, a_1$ must be followed on the operand stack by a reference r to an (initialized) object in the heap h
- the class $id_c{}'''$ of the referenced object must be the same as $id_c{}'$, or a subclass thereof
- class $id_c{}'''$ and its superclasses are searched for the declaration and implementation of a method with signature sig', using the function *lookup* (see p. 13); this should yield the name $id_c{}''''$ of the class implementing the method, and the access modifiers acc_m, return descriptor d', local variable limit n_l, and code array $code'$ of the method
- the current class must have access to the class $id_c{}''''$ implementing the method
- the method must not be declared (by class $id_c{}''''$) to be `private`, `static`, `abstract`, or `native`
- if the method is declared to be `protected`, then the current class must be the same as class $id_c{}'$, or a subclass thereof, and the class $id_c{}''''$ implementing the method must be the same as the current class, or a subclass thereof
- the declared return descriptor d' must be the same as that of the symbolic method reference (i.e., d)
- the new frame f' is based on the code array $code'$ and class name $id_c{}''''$ of the method implementation found by *lookup*
- the reference r is passed as the first argument to the invoked method, i.e., in local variable 0 of frame f'.

It is assumed that the `invokevirtual` instruction can also be used for invoking an instance method of class `java.lang.Object` for an array object (cf. Section 3.3.6):

$$\frac{\begin{array}{l} instr(ts) = \texttt{invokevirtual}\ i \\ pool(ts)(i) = (id_c{}', sig', d) : Const_m \\ id_c{}' = \texttt{java.lang.Object} \\ e(id_c{}') = ((acc_c, id_c{}'', is, fd, cv, md, mi, cp), sv) \\ sig' = (id_m, <d_1, d_2, d_3, \ldots, d_k>) \\ id_m \notin \{\texttt{<init>}, \texttt{<clinit>}\} \\ md(sig') = (acc_m, d', excs) \\ acc_m \cap \{\texttt{private}, \texttt{static}, \texttt{abstract}, \texttt{native}\} = \emptyset \\ d = d' \\ mi(sig') = (n_s, n_l, code', hdls') \\ s = as@sr \\ as = <a_k, a_{k-1}, a_{k-2}, \ldots, a_2, a_1>@<r : Ref> \\ size(as) \leq n_l \\ \forall 1 \leq j \leq k.(initialized(a_j, h) \wedge compatVal(d_j, a_j, h, e)) \\ h(r) = (d'', k', av) : Obj_a \\ (min_{pc}(code'), <>, args(as), (id_c{}'''', sig')) = f' : Frame \end{array}}{ts \Rightarrow (f' :: (pc, sr, l, m) :: fr, h, e)} \quad (50)$$

Again, the semantics of the `invokevirtual` instruction, when used in this way, is similar to that of the `invokestatic` instruction above (rule 48). However,

- the constant pool entry at index i must be a symbolic method reference, referring to a method in class `java.lang.Object`
- the method must not be declared to be `private` or `static`
- the method arguments $a_k, \ldots, a_1$ must be followed on the operand stack by a reference r to an array object in the heap h
- the reference r is passed as the first argument to the invoked method.

The instruction `invokeinterface` invokes an interface method:

$$\frac{\begin{array}{l} instr(ts) = \texttt{invokeinterface}\ i\ n \\ pool(ts)(i) = (id_c{}', sig', d) : Const_{im} \\ e(id_c{}') = ((acc_c, id_c{}'', is, fd, cv, md, mi, cp), sv) \\ \texttt{interface} \in acc_c \\ sig' = (id_m, <d_1, d_2, d_3, \ldots, d_k>) \\ id_m \notin \{\texttt{<init>}, \texttt{<clinit>}\} \\ sig' \in dom(md) \\ s = as@sr \\ as = <a_k, a_{k-1}, a_{k-2}, \ldots, a_2, a_1>@<r : Ref> \\ size(as) = n \\ \forall 1 \leq j \leq k.(initialized(a_j, h) \wedge compatVal(d_j, a_j, h, e)) \\ h(r) = (id_c{}''', iv) : Obj_c \\ id_c{}' \in interfaces(id_c{}''', e) \\ lookup(sig', id_c{}''', e) = (id_c{}'''', acc_m, d', n_l, code') \\ access(id_c{}'''', id_c, e) \\ acc_m \cap \{\texttt{public}, \texttt{static}, \texttt{abstract}, \texttt{native}\} = \{\texttt{public}\} \\ d = d' \\ size(as) \leq n_l \\ (min_{pc}(code'), <>, args(as), (id_c{}'''', sig')) = f' : Frame \end{array}}{ts \Rightarrow (f' :: (pc, sr, l, m) :: fr, h, e)} \quad (51)$$

The semantics of the `invokeinterface` instruction is similar to that of the `invokevirtual` instruction above (rule 49). However,

- the `invokeinterface` instruction takes an extra immediate operand $n \in \mathcal{N}$
- the constant pool entry at index i must be a symbolic interface method reference, refererring to a method in interface $id_c{}'$ with signature sig' and return type descriptor d
- the total size of the method arguments as must equal the immediate operand n
- the class $id_c{}'''$ of the object referenced by stack operand r must implement interface $id_c{}'$
- the method retrieved by *lookup* must be declared to be `public`, but not `static`, `abstract`, or `native`.

The instruction `invokespecial` can be used for invoking a 'special' instance method:

$$\frac{\begin{array}{l} instr(ts) = \texttt{invokespecial}\ i \\ pool(ts)(i) = (id_c{}', sig', d) : Const_m \\ id_c \neq id_c{}' \ \wedge \ id_c{}' \in supers(id_c, e) \\ e(id_c{}') = ((acc_c, id_c{}'', is, fd, cv, md, mi, cp), sv) \\ acc_c \cap \{\texttt{interface}, \texttt{super}\} = \{\texttt{super}\} \\ sig' = (id_m, <d_1, d_2, d_3, \ldots, d_k>) \\ id_m \notin \{\texttt{<init>}, \texttt{<clinit>}\} \\ sig' \in dom(md) \\ lookup(sig', super(id_c, e), e) = (id_c{}'''', acc_m{}', d', n_l, code') \\ access(id_c{}'''', id_c, e) \\ acc_m \cap \{\texttt{private}, \texttt{static}, \texttt{abstract}, \texttt{native}\} = \emptyset \\ d = d' \\ s = as@sr \\ as = <a_k, a_{k-1}, a_{k-2}, \ldots, a_2, a_1>@<r : Ref> \\ size(as) \leq n_l \\ \forall 1 \leq j \leq k.(initialized(a_j, h) \wedge compatVal(d_j, a_j, h, e)) \\ h(r) = (id_c{}''', iv) : Obj_c \\ id_c{}' \in supers(id_c{}''', e) \\ \texttt{protected} \in acc_m \Rightarrow (id_c{}' \in supers(id_c, e) \ \wedge \ id_c \in supers(id_c{}''', e)) \\ (min_{pc}(code'), <>, args(as), (id_c{}'''', sig')) = f' : Frame \end{array}}{ts \Rightarrow (f' :: (pc, sr, l, m) :: fr, h, e)} \quad (52)$$

The semantics of the `invokespecial` instruction is similar to that of the `invokevirtual` instruction above (rule 49). However,

- the symbolic method reference must refer to a method in a superclass of the current class
- the `super` flag must be set for class $id_c{}'$
- the method must not be declared (by class $id_c{}'$) to be `private` or `static`
- the superclasses of the current class, rather than those of the class of the object, are searched for the declaration and implementation of the method to be invoked.

The `invokespecial` instruction can also be used for invoking a constructor, a `private`

instance method, or a method of the current class:

$$\frac{\begin{array}{l} instr(ts) = \texttt{invokespecial}\ i \\ pool(ts)(i) = (id_c{}', sig', d) : Const_m \\ e(id_c{}') = ((acc_c, id_c{}'', is, fd, cv, md, mi, cp), sv) \\ \texttt{interface} \notin acc_c \\ access(id_c{}', acc_c, id_c) \\ sig' = (id_m, <d_1, d_2, d_3, \ldots, d_k>) \\ id_m \neq \texttt{<clinit>} \\ md(sig') = (acc_m, d', excs) \\ id_m = \texttt{<init>} \ \vee\ \texttt{private} \in acc_m \ \vee\ id_c = id_c{}' \ \vee\ id_c{}' \notin supers(id_c, e) \ \vee\ \texttt{super} \notin acc_c \\ acc_m \cap \{\texttt{static}, \texttt{abstract}, \texttt{native}\} = \emptyset \\ \texttt{private} \in acc_m \Rightarrow id_c = id_c{}' \\ mi(sig') = (n_s, n_l, code', hdls') \\ s = as@sr \\ as = <a_k, a_{k-1}, a_{k-2}, \ldots, a_2, a_1>@<r : Ref> \\ size(as) \leq n_l \\ \forall 1 \leq j \leq k.(initialized(a_j, h) \wedge compatVal(d_j, a_j, h, e)) \\ h(r) = o : (Obj_u \cup Obj_c) = (id_c{}''', iv) \\ (id_m = \texttt{<init>} \wedge o \in Obj_u) \ \vee\ (id_m \neq \texttt{<init>} \wedge o \in Obj_c) \\ id_c{}' \in supers(id_c{}''', e) \\ \texttt{protected} \in acc_m \Rightarrow (id_c{}' \in supers(id_c, e) \ \wedge\ id_c \in supers(id_c{}''', e)) \\ (min_{pc}(code'), <>, args(as), (id_c{}', sig')) = f' : Frame \\ initObj(id_c{}', sig', r, h) = h' \end{array}}{ts \Rightarrow (f' :: (pc, sr, l, m) :: fr, h', e)} \quad (53)$$

The semantics of the `invokespecial` instruction, when used in this way, is similar to that of the `invokestatic` instruction above (rule 48). However,

- the name of the method may be `<init>`
- the method must not be declared to be `static`
- the method arguments $a_k, \ldots, a_1$ must be followed on the operand stack by a reference r to a class instance in the heap h
- if the referenced object is uninitialized, then the method to be invoked must be a constructor (named `<init>`); if the object has already been initialized, the method must not be a constructor
- the class $id_c{}'''$ of the object must be the same as $id_c{}'$, or a subclass thereof
- if the method is declared to be `protected`, then the current class must be the same as class $id_c{}'$, or a subclass thereof, and the class $id_c{}''''$ implementing the method must be the same as the current class, or a subclass thereof.

Note that the utility function *initObj* (see p. 15) is used for tagging the object referenced by the stack operand r as initialized, in case the method being invoked is the constructor of class `java.lang.Object`; hence, further instructions (including those of the `java.lang.Object` constructor) may refer to members of the object.

2.5.14 Normal Method Return

The instruction `ireturn` returns from the current method, and passes a value of type *Int* to the invoker:

$$\frac{\begin{array}{l} instr(ts) = \texttt{ireturn} \\ e(id_c) = ((acc_c, id_c{}', is, fd, cv, md, mi, cp), sv) \\ md(sig) = (acc_m, d, excs) \\ d \in \{\texttt{boolean}, \texttt{char}, \texttt{byte}, \texttt{short}, \texttt{int}\} \\ s = (k : Int) :: sr \\ fr = (pc', s', l', m') :: fr' \\ (fr, h, e) = ts' : TS \\ size(s') + size(k) \leq max_s(ts') \\ succ(ts') = pc'' \end{array}}{ts \Rightarrow ((pc'', k :: s', l', m') :: fr', h, e)} \quad (54)$$

If it holds that

- the current method is declared (by the current class) to have return type `boolean`, `char`, `byte`, `short`, or `int`
- the topmost stack operand k is of type *Int*
- the frame stack holds at least one more frame besides the current frame (namely, the frame of the invoker of the current method)
- the operand stack of the invoker does not overflow by pushing a value of type *Int* onto it
- the address of the next instruction in the invoking method is in the set of valid target addresses of the invoker

then the current frame is popped off the frame stack, the value k is pushed onto the operand stack of the invoker, and execution continues with the next instruction of the invoker.

It is assumed that a value of type *Int* must not be truncated when it is passed as return value from a method declared to have return type `boolean`, `char`, `byte`, or `short` (cf. Section 3.3.7).

Note that the proviso $succ(ts') = pc''$ in the above transition rule corresponds to a dynamic verification (at run-time) that

> *Execution never falls off the bottom of the code array.*
> (*The Java Virtual Machine Specification*[2, p. 123])

If a similar proviso had instead occurred in the premises of each of the method invocation instructions (rule 48 ff), this would have corresponded to a static (load-time) verification of the same property.

The following instructions also return from the current method, and pass the topmost stack operand to the invoker; the semantics of these instructions is very similar to that of the `ireturn` instruction above (rule 54):

- `lreturn` returns a value of type *Long*, provided the declared return type is `long`
- `freturn` returns a value of type *Float*, provided the declared return type is `float`
- `dreturn` returns a value of type *Double*, provided the declared return type is `double`

The instruction `areturn` returns from the current method, and passes an object reference (or the special value `null`) to the invoker:

$$\frac{\begin{array}{l} instr(ts) = \texttt{areturn} \\ e(id_c) = ((acc_c, id_c{}', is, fd, cv, md, mi, cp), sv) \\ md(sig) = (acc_m, d, excs) \\ d \notin Simple \\ s = (r : Ref_0) :: sr \\ initialized(r, h) \\ compatVal(d, r, h, e) \\ fr = (pc', s', l', m') :: fr' \\ (fr, h, e) = ts' : TS \\ size(s') + size(r) \leq max_s(ts') \\ succ(ts') = pc'' \end{array}}{ts \Rightarrow ((pc'', r :: s', l', m') :: fr', h, e)} \quad (55)$$

The semantics of the `areturn` instruction is similar to that of the `ireturn` instruction above (rule 54). However, the return type must not be a *Simple* type, the topmost stack operand r must not be a reference to an uninitialized object in the heap, and r must be assignment compatible with the declared return type of the current method.

The instruction `return` returns from the current method, without passing any value to the invoker:

$$\frac{\begin{array}{l} instr(ts) = \texttt{return} \\ e(id_c) = ((acc_c, id_c{}', is, fd, cv, md, mi, cp), sv) \\ md(sig) = (acc_m, \texttt{none}, excs) \\ fr = (pc', s', l', m') :: fr' \\ succ((fr, h, e) : TS) = pc'' \end{array}}{ts \Rightarrow ((pc'', s', l', m') :: fr', h, e)} \quad (56)$$

The semantics of the `return` instruction is similar to that of the `ireturn` instruction above (rule 54), except that the current method must be declared to return no value, the contents of the operand stack is discarded, and no value is pushed onto the operand stack of the invoker.

2.5.15 Native Method Invocation

A method may be declared to be `native`, meaning that the method is not implemented via Java byte code, but via some implementation specific form of code. The class declaring the method as `native` is considered to implement the method, although the method is not supposed to be in the set of method implementations of the class.

The method invocation instructions specified in Section 2.5.13 can also be used for invoking

native methods, e.g.

$$\frac{\begin{array}{l} instr(ts) = \texttt{invokestatic}\ i \\ pool(ts)(i) = (id_c{}', sig', d) : Const_m \\ e(id_c{}') = ((acc_c, id_c{}'', is, fd, cv, md, mi, cp), sv) \\ \texttt{interface} \notin acc_c \\ access(id_c{}', acc_c, id_c) \\ sig' = (id_m, <d_1, d_2, d_3, \ldots, d_k>) \\ id_m \notin \{\texttt{<init>}, \texttt{<clinit>}\} \\ md(sig') = (acc_m, d', excs) \\ acc_m \cap \{\texttt{static}, \texttt{abstract}, \texttt{native}\} = \{\texttt{static}, \texttt{native}\} \\ \texttt{private} \in acc_m \Rightarrow id_c = id_c{}' \\ \texttt{protected} \in acc_m \Rightarrow id_c{}' \in supers(id_c, e) \\ d = d' \\ s = as@sr \\ as = <a_k, a_{k-1}, a_{k-2}, \ldots, a_2, a_1> \\ \forall 1 \leq j \leq k.(initialized(a_j, h) \wedge compatVal(d_j, a_j, h, e)) \\ invokeNative(ts) = ts' : TS \end{array}}{ts \Rightarrow ts'} \quad (57)$$

The semantics of the `invokestatic` instruction, when used for invoking a native method, is similar to that specified in rule 48. However,

- the method must be declared to be `native`
- class $id_c{}'$ should not implement the method (via the *mi* component of the class declaration)
- there is no restriction on the total size of the list of method arguments *as*
- instead of pushing a new frame onto the frame stack, the abstract semantic function *invokeNative* is used for retrieving the implementation specific code of the invoked method and execute it in an implementation specific fashion.

Similarly, the `invokevirtual`, `invokeinterface`, and `invokespecial` instructions can also be used for invoking native methods. The semantics of these instructions, when used on native methods, differ from that specified in Section 2.5.13 in the same way as described above for the `invokestatic` instruction.

The Java Virtual Machine Specification[2] does not specify which parts of the Java Virtual Machine state a native method has access to. Since native methods must generally be able to interact with the Java Virtual Machine (cf. Section 3.2), it is assumed that native methods have unlimited access to the entire internal state of the Java Virtual Machine, and may change it in any conceivable way.

Note that it is not specified how a native method returns to the invoker, nor how a return value should be passed to the invoker; again, this is not described by *The Java Virtual Machine Specification*[2].

2.5.16 Class and Interface Initialization

The instructions for instantiating a class, and for accessing the members of a class or interface, require the class or interface to be already loaded and initialized. If the declaration of the class

or interface is not already in the environment e of the thread state ts, the class or interface must first be loaded from the global environment and initialized. This is considered to precede the execution of the instruction causing the class or interface to be loaded, e.g.

$$\frac{\begin{array}{l} instr(ts) = \texttt{new}\ i \\ pool(ts)(i) = id_c{}' : Id_c \\ id_c{}' \notin dom(e) \\ loadClass(id_c{}', h, e) = (cs, h' : Heap, e' : Env) \\ cs = <c_1, c_2, c_3, \ldots, c_n> : (Id_c \times Code)^* \\ wf_e(e) \Rightarrow wf_e(e') \\ (\texttt{<clinit>}, <>) = sig' : Sig \\ \forall 1 \leq j \leq n.(c_j = (id_j, code_j) \ \wedge \ (min_{pc}(code_j), <>, \emptyset, (id_j, sig')) = f_j) \\ <f_1, f_2, f_3, \ldots, f_n> = fs : Frame^* \end{array}}{ts \Rightarrow (fs@(f :: fr), h', e')} \quad (58)$$

If it holds that

- the class or interface to be instantiated by the `new` instruction is not declared in the environment e
- the declaration of the class or interface, and all superclasses of that class or interface, can be loaded from the global environment, using the utility function *loadClass* (see p. 2.4.2)

then the frames corresponding to the initializer methods of the newly loaded classes/interfaces are pushed onto the frame stack.

Note that the list of initializer methods cs returned by the *loadClass* function holds the class names and code arrays for the `<clinit>` methods of the newly loaded classes/interfaces declaring such methods. The order of the `<clinit>` methods in the list ensures that the superclasses of a class or interface are initialized before the class/interface itself.

Note also, that the `<clinit>` methods are assumed to be `static`, but not `abstract`; other access flags of the `<clinit>` methods are ignored. Furthermore, the `<clinit>` methods are assumed to take no parameters, and to return no values (i.e., to have return type `void`).

Similarly to the `new` instruction above, the following instructions can also cause the loading and initialization of a class or interface: `putstatic`, `getstatic`, `invokestatic`, `invokeinterface`, `invokevirtual`, and `invokespecial`. The semantics of these instructions, concerning loading and initialization of a class or interface, is very similar to that specified above (rule 58).

2.5.17 Exceptions

The instruction `athrow` can be used for explicitly throwing an exception:

$$\frac{\begin{array}{l} instr(ts) = \texttt{athrow} \\ s = (r : Ref) :: sr \\ h(r) = o : Obj_c \\ instOf(\texttt{java.lang.Throwable} : Id_c, o, e) \\ throw(r, ts) = ts' : TS \end{array}}{ts \Rightarrow ts'} \quad (59)$$

If it holds that the topmost stack operand is a reference to an initialized instance of class `java.lang.Throwable` (according to the *instOf* predicate; see p. 18), then the utility function

throw (see p. 16) is used for throwing the referenced object in the current frame. If the exception is not handled within the current frame, it is rethrown in the frame of the invoker, etc.

Note that the thrown object does not have to be an instance of any exception class that the current method is declared to throw.

2.5.18 Type Check

The instruction `checkcast` verifies that an object can be 'cast' to a specified class, interface or array type:

$$\frac{\begin{array}{l} instr(ts) = \texttt{checkcast}\ i \\ pool(ts)(i) = d : (Array \cup Id_c) \\ s = (r : Ref_0) :: sr \\ r \neq null \Rightarrow (r \in dom(h) \wedge instOf(d, h(r), e)) \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', s, l, m) :: fr, h, e)} \quad (60)$$

The `checkcast` instruction takes an immediate operand $i \in Index$. If it holds that

- i is a valid index into the constant pool of the current class
- the constant pool entry at index i is a symbolic class or interface reference, or an array type descriptor
- the topmost stack operand r is the special value `null`, or a reference to an instance of the class, interface or array type specified by d (according to the *instOf* predicate; see p. 18)

then the topmost stack operand is popped, and execution continues with the next instruction in the current method (under the standard assumptions).

It is assumed that the current class need not have permission to access the specified class or interface type, or the base type of the specified array type.

The instruction `instanceof` also examines the type of an object:

$$\frac{\begin{array}{l} instr(ts) = \texttt{instanceof}\ i \\ pool(ts)(i) = d : (Array \cup Id_c) \\ s = (r : Ref_0) :: sr \\ instOf_i(d, r, h, e) = k : Int \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', k :: sr, l, m) :: fr, h, e)} \quad (61)$$

The semantics of the `instanceof` instruction is similar to that of the `checkcast` instruction above (rule 60). However, the utility function $instOf_i$ (see p. 13) is used for checking the type of the object against the specified class, interface, or array type. The value k returned by $instOf_i$ is 1 or 0, indicating whether or not the object is an instance of the specified class, interface, or array type.

The topmost stack operand is popped, the value k is pushed onto the operand stack, and execution continues with the next instruction in the current method (under the standard assumptions). Note that $size(k) \leq size(r)$ is assumed to hold.

Note also, that the above type checking instructions treat the special value `null` differently: for `checkcast`, `null` is treated as having the specified type; for `instanceof`, `null` is not treated as an instance of the specified class, interface, or array type.

2.5.19 Miscellanous

The instruction `nop` does nothing:

$$\frac{\begin{array}{l} instr(ts) = \texttt{nop} \\ succ(ts) = pc' \end{array}}{ts \Rightarrow ((pc', s, l, m) :: fr, h, e)} \quad (62)$$

Provided the address of the next instruction in the current method is in the set of valid target addresses for the thread state ts, execution proceeds with the next instruction of the current method.

The instructions `monitorenter` and `monitorexit` have not been specified, since these instructions are only relevant in connection with multi-threading. As long as multi-threading is not considered, these instructions can be regarded as having the same semantics as the `pop` instruction (rule 11).

2.6 Error Semantics

In the previous sections, the 'normal' operation of each of the Java Virtual Machine instructions has been specified. It has, however, not been specified what is supposed to happen in case the premises of one or more transition rules are not satisfied, so that none of the rules apply.

For example, if a method attempts to perform a division by zero, using the `idiv` instruction, an instance of exception class `java.lang.ArithmeticException` must be thrown.

Note that there is a large number of error conditions in which a specific exception must be thrown; *The Java Virtual Machine Specification*[2] describes most of these in detail. At the abstract level of this specification, however, error conditions are not differentiated between.

Note also, that an exception thrown due to an error condition might not be handled within any of the frames on the frame stack. In such a situation, the execution of the Java Virtual Machine is considered to terminate abruptly due to an unhandled exception.

2.7 Virtual Machine Startup and Termination

When the Java Virtual Machine starts executing, it initially attempts to invoke the method `main` in a specified class. That method must be declared to be `public` and `static`, it must return no value, and it must accept an array object of component type `java.lang.String` as argument.

The method is invoked as if by execution of instruction `invokestatic` (see rule 48). This implies that the class implementing method `main`, and all of its superclasses, will be loaded from the global environment and initialized.

Method `main` may, in turn, access members of other classes and interfaces, thereby causing those classes and interfaces to be loaded and initialized, etc.

If method `main` returns, it must be via execution of the `return` instruction. However, rule 56, specifying the semantics of that instruction, does not apply in the case of return from method `main`, since there will be no more frames on the frame stack at this point. This

situation, although in principle an error condition, is considered normal termination of the Java Virtual Machine.

As noted in the previous section, the execution of the Java Virtual Machine may also terminate abruptly due to an unhandled exception being thrown.

Furhermore, the execution of the Java Virtual Machine may terminate due to invocation of the method `exit` of class `java.lang.Runtime`, or method `exit` in class `java.lang.System`; this is, however, beyond the scope of this specification.

3 Ambiguities in Sun's Specification

In the course of developing the formal specification presented in the previous sections, we have found *The Java Virtual Machine Specification*[2] to be significantly less precise than desirable for a complete, unambiguous specification.

This section describes the issues on which Sun's informal specification is deemed to be too inaccurate, and explains the corresponding assumptions underlying the formal specification.

See also Appendix A for a list of minor errors in *The Java Virtual Machine Specification*[2].

3.1 Implementation of Special Java Classes

The library of 'standard' classes defined in the Java programming language plays a central rôle in the semantics of the language. Some of the most important ones are these classes in the package `java.lang`:

`Object` This is the superclass of all other classes; it has no superclasses. The special method `clone` can be used for creating a copy of any instance of a class that implements the interface `Cloneable`. Class `Object` also implements the special methods `hashCode`, and `getClass`.

`Class` Instances of this class are assumed to be created for every class or interface loaded into the Java Virtual Machine. The special method `newInstance` can be used for creating a new instance of any non-abstract class.

`ClassLoader` This abstract class provides an alternative means of loading class files into the Java Virtual Machine. The special method `defineClass` can be used for converting an array of bytes into a an instance of class `Class`. The special method `resolveClass` can be used for resolving a `Class` object.

`String` All strings in Java are instances of this class. String literals are assumed to be initialized as 'internalized' (i.e., unique) `String` objects; the special instance method `intern` can be used for 'internalizing' any `String` object.

`Throwable` Any exception thrown due to an error, or explicitly from within a Java method, must be an instance of this class. The special method `fillInStackTrace` is used for taking a 'snapshot' of the current call stack, and storing this in the exception object.

`Runtime` A Java application will always have access to a single instance of class `Runtime`; this object serves as an interface to the Java Virtual Machine. Class `Runtime` implements a number of special methods for retrieving information about the amount of available memory, for starting other system processes, for invoking the garbage collector, etc.

`System` This class implements a number of static methods for accessing the current state of the Java Virtual Machine, for interacting with the environment in which the Java Virtual Machine is executing, etc.

The semantics of the above classes cannot be isolated completely from the semantics of the Java Virtual Machine. Other standard Java classes also depend heavily on interaction with the Java Virtual Machine, e.g. many of the classes in package `java.io`.

Furthermore, the operation of the Java Virtual Machine depends on the existence and semantics of a number of Java classes, e.g. `java.lang.Object`, `java.lang.Class`, `java.lang.String`, and `java.lang.Throwable`.

Unfortunately, *The Java Virtual Machine Specification*[2] does not specify in which way, or to what extent, an implementation of the Java Virtual Machine must be integrated with the implementation of those special Java classes.

It is also not described which Java classes must be available to the Java Virtual Machine (presumably, those in the packages `java.lang`, `java.io`, and `java.util`), or which classes must be considered 'already loaded' at virtual machine startup (presumably, none).

The normal procedure for (loading, linking, and) initialization of classes and interfaces [2, p. 46] presents some serious problems in connection with initialization of the special classes `java.lang.Object` and `java.lang.Class`: before class `java.lang.Object` can be initialized, class `java.lang.Class` must be initialized, in order to create a `Class` object representing class `Object` during its initialization.

However, initialization of class `java.lang.Class` requires that its superclass, namely, `java.lang.Object`, has first been initialized; hence this is not possible. Similarly, a `Class` object would have to be created for representing class `java.lang.Class` during its initialization; however, the `Class` object cannot be created before the class has been initialized.

The Java Virtual Machine Specification[2] does not describe the above problems relating to initialization of class `java.lang.Object` and `java.lang.Class`, nor does it devise any other initialization procedure to be used for these classes.

3.2 Interface to Native Methods

Many standard Java classes cannot be implemented solely via Java byte code; such classes must be implemented via native methods, or be supported/implemented directly by the Java Virtual Machine. Unfortunately, this is not described in *The Java Virtual Machine Specification*[2], although the implementation of a number of standard Java classes must be integrated with the Java Virtual Machine.

Furthermore, *The Java Virtual Machine Specification*[2] does not describe which parts of the Java Virtual Machine state native methods are permitted to access. Presumably, native methods must have unlimited access to the entire internal state of the Java Virtual Machine.

It is also not specified how native methods are supposed to return to the invoker, nor how a return value should be passed to the invoker.

3.3 Unspecified Details

The following sections list various unspecified details and contradictions in *The Java Virtual Machine Specification*[2].

3.3.1 Class File Verification

It is not clear whether or not

- multiple declarations of a method (with the same signature) in a class file is permitted
- multiple declarations of a field in a class file is permitted
- a class file is permitted to declare multiple `<clinit>` methods (with different signatures)

- the description of verification of class files in 4.9 is to be considered a (general) specification of class file verification, or an explanation of Sun's implementation of the Java Virtual Machine
- the last instruction in the code array for a method can be `jsr` or `jsr_w` (maybe this should be caught by a static verification that "execution never falls off the bottom of the code array")
- the last instruction in the code array of a method can be a method invocation instruction
- the max_stack item of a Code attribute for a method must be greater than zero in case the attribute also declares one or more exception handlers

3.3.2 Class and Interface Initialization

It is not clear when interfaces are to be loaded, prepared, and initialized. The approach of initializing at the first "active use" of an interface, i.e., in connection with access to a member of the interface, does not suffice.

For example, a situation might occur where the type of an object must be compared to the type of a field, before storing a reference to the object into the field. In case the field is declared to have an interface type, the superinterfaces of the class of the object must be searched, in order to determine if the class of the object implements the interface.

Had the field been declared to have a class type instead, then the superclasses of the object could readily have been searched (since these must have been loaded and initialized when one of their subclasses has been instantiated) to determine if the object is an instance of the class type of the field.

Furthermore, it is not clear

- how a ConstantValue field attribute, when appearing in the declaration of a non-`final`, `static` field should be used
- whether or not there are any restrictions on the order of initializations within the `<clinit>` method of a class
- how class `java.lang.String` is supposed to be instantiated in connection with the `ldc` and `ldc_w` instructions, and in connection with class/interface initialization.

3.3.3 Class and Array Instantiation

It is not clear

- what the semantics of the `anewarray` and `newarray` instructions is, in case the count operand is zero; presumably, a new, empty, and distinct array object must be created
- what the semantics of the `multianewarray` instruction is, in case one of the count stack operands is zero; presumably, all of the following counts are to be ignored
- why *The Java Virtual Machine Specification*[2, p. 317] specifies that the current class must "... have permission to access the base class of the resolved array class ...", in connection with the `multianewarray` instruction; presumably, this is not necessary

- whether or not the superclass constructor `java.lang.Object.<init>` must be invoked during initialization of an array object (presumably, no).

3.3.4 Array Access

It is not clear

- what the semantics of the `bastore` and `baload` instructions is, when these instructions are used to access a component of a `boolean` array
- how the *Int* stack operand of the `bastore` instruction is to be truncated/transformed
- how a component in a boolean array should be converted to an *Int* result by the `baload` instruction

3.3.5 Field Access

It is not clear

- whether or not a `final` field can be stored into by the `putstatic` and `putfield` instructions (presumably not)
- whether or not an interface field can be loaded by the `getstatic` instruction (presumably, yes)
- what the semantics of the `putfield` and `putstatic` instructions is, if the field descriptor type is `boolean` (presumably, the same as for the other 'small' Java types)
- how the *Int* stack operand is to be converted/truncated, when the `putfield` or `putstatic` instruction is used for storing a value into a `boolean`, `byte`, `char`, or `short` field (presumably, no conversion/truncation)
- how the value of a field of type `boolean`, `byte`, `char`, or `short` is to be extended to a value of type *Int*, when the `getfield` and `getstatic` instructions are used for loading the value of the field
- whether or not a symbolic field reference (a CONSTANT_FieldRef) can refer to a method of an array class (presumably, members of class `java.lang.Object` must be referred to instead)

3.3.6 Method Invocation

It is not clear

- what the semantics of the `invokespecial` instruction is, if the ACC_SUPER access flag is not set for the current class (or, more generally, if the four criteria on p. 261 are not met)
- how an actual *Int* parameter is to be converted/truncated to a corresponding formal parameter of type `boolean`, `byte`, `char`, or `short`, in connection with method invocation (presumably, no conversion/truncation)

- whether or not a method reference (a CONSTANT_Methodref) can refer to a method of an array class (presumably, methods in class `java.lang.Object` must be referred to instead; that is, arrays implicitly inherits methods from `java.lang.Object`, but the actual invocation of these must refer directly to class `java.lang.Object`, and must use instruction `invokevirtual`.

3.3.7 Normal Method Return

It is not clear

- what the resulting program counter is, after "reinstating the invoker" in connection with all of the method return instructions
- whether or not the `ireturn` instruction must be used for returning a value of type `boolean` (presumably, yes)

3.3.8 Uninitialized Class Instances

It is not clear

- if the `ret` instruction is considered a (possibly) "backwards branch" with respect to uninitialized class instances (p. 122)
- if the `areturn` instruction can be used to return (a reference to) an uninitialized class instance to the invoker of the method
- if the `athrow` instruction can be used to throw (a reference to) an uninitialized class instance
- if the `checkcast` and `instanceof` instructions can be used on (a reference to) an uninitialized class instance
- if (a reference to) an uninitialized class instance be stored into a field, using the `putfield` and `putstatic` instructions, or if such a value can be loaded from a field using `getfield` and `getstatic` instructions
- if the stack manipulation instructions can be used on references to uninitialized class instances

3.3.9 Miscellaneous

It is not clear

- how overflow is to be treated in connection with the `iinc` instruction (presumably, overflow should be ignored)
- if overflow can occur during the offset calculation in connection with the `tableswitch` instruction (presumably not)
- if overflow can occir in connection with instruction address calculations (presumably not)
- what the semantics of the `aastore`, `checkcast`, and `instanceof` instructions is, in case the source type (S) is an array type, and the target type (T) is an interface type

3.4 Specification in Terms of Java Concepts

- In a number of places, the semantics of exceptions at the level of the JVM are explained in terms of Java concepts, rather than Java Virtual Machine concepts. For example, the term "a finally clause" is used in the explanation of structural constraints on Java byte code concerning uninitialized class instances (p. 122), although this could be explained more precisely in terms of exception handlers (at the level of the Java Virtual Machine).

- The description of the array operations (Taload, Tastore) explains that an exception is thrown "if index is not within the bounds of the array ...". It is not explained precisely what this means at the level of the JVM, i.e., that array indices are zero-based, and that valid indices are 0 through $n-1$, where n is the length of the array object. (This follows from the semantics of Java, but is not specified at the level of the Java Virtual Machine.)

- The explanation of restrictions in connection with `jsr`, `jsr_w` and `ret` instructions (p. 124) is not very clear, nor does it appear to be very precise. For example, it is not clear how the instruction following a `jsr` instruction be returned to by more than one `ret` instruction.

3.5 Implementation Specific Details

In a few places, the general description of the Java Virtual Machine refers to Sun's implementation of the Java Virtual Machine:

- In the explanation of the getfield and putfield instructions (p. 226 and 326), it is claimed that resolving a field reference leads to a "field offset" into the class instance.

- The explanation of the `invokeinterface`, `invokespecial`, `invokestatic`, and `invokevirtual` instructions (p. 258 ff) mentions a "method table", "method table entry", and "direct reference to the code for ...".

- It is mentioned that the constant pool can contain dynamic data (p. 262 and 265): "The constant pool entry representing the resolved method ...".

- The description mentions the "nargs" value for a resolved method (p. 262, 265, and 267) and the "index into the method table of the resolved class" (p. 267).

- On p. 133 it is mentioned that occurrences of an uninitialized class instance can be held "in registers", although this term is not explained.

4 Conclusion

A formal, abstract specification of the semantics of the Java Virtual Machine, and thereby of Java byte Code, has been developed.

During the development of the formal specification, a number of errors and ambiguities in the informal, official specification found in *The Java Virtual Machine Specification*[2] have been uncovered. These have been described, and the authors of the book have been made aware of the problems.

Furthermore, the assumptions underlying the formal specification, due to ambiguities in *The Java Virtual Machine Specification*[2], have been explained.

The formal specification of the Java Virtual Machine semantics could be improved in a number of ways, e.g.

- the abstract functions and predicates presented in Section 2.4.1 could be specified more precisely and exhaustively, in particular with respect to arithmetic operations, conversion between and truncation of simple types, and comparison of values of simple types
- the semantics the Java Virtual Machine instructions with respect to errors is treated only superficially; it could be specified precisely which exception should be raised in each possible error condition
- the semantics of multi-threading in the Java Virtual Machine could be specified
- ambiguous parts of the semantics of the Java Virtual Machine could be specified, provided *The Java Virtual Machine Specification*[2] was improved in this respect.

A Errata in Sun's Specification

This appendix lists errata in *The Java Virtual Machine Specification*[2] that has been discovered in the course of developing the formal specification of the Java Virtual Machine semantics; in other words, the following list includes details in the book that are obviously wrong. Some of the more subtle problems in Sun's specification of the Java Virtual Machine are described in Section 3.

Page numbers in the following list refer to *The Java Virtual Machine Specification*[2].

The list of errata has been sent by e-mail to jvm@java.sun.com, and will probably appear on the Web-page listing the 'official' set of errata for the book sometime in the near future.

p. xv The very first sentence of the preface reads:

> *This book has been written as a complete specification for the Java Virtual Machine.*

Presumably, this has been the original intention of the authors. However, after studying the book rather closely, it must be concluded that it is not a complete specification of the Java Virtual Machine, since vital parts of its semantics are not specified, e.g. the interaction with special Java classes, and the interface to native methods (cf. Section 3).

p. 47 Concerning the detailed procedure for initialization of a class or interface, it is assumed that

> *... the Class object has already been verified and prepared, ...*

Presumably, it is the *class file* that must already have been verified and prepared (prior to initialization proper).

Furthermore, the sentence quoted above contains a minor grammatical error:

> *... and that the Class object contains state that can indicates one of four situations ...*

p. 72 In the description of table 3.1, it is claimed that

> *Only instructions that exist for multiple types are listed.*

However, the `iinc` and `lcmp` instructions are the only instructions in the rows for `Tinc` and `Tcmp`, respectively.

p. 81 The section heading

> *3.11.9 Throwing and Handling Exceptions*

is sligtly misleading; that section does not describe how exceptions are handled.

p. 86 In the description of restrictions on combinations of access flags in a class file, it is stated that

> *An interface cannot be final; its implementation could never be completed if it were ...*

Presumably, an interface could very well be flagged as ACC_FINAL, and then also be implemented by some class.

p. 88 Concerning the interfaces item of a class file, it is stated that (the indices of constant pool entries representing) the direct superinterfaces of the class/interface must appear

> *... in the left-to-right order given in the source for the type.*

Presumably, the direct superinterfaces of the class or interface may appear in any order.

p. 88 In the description of the methods item of a class file, it is stated that

> *Each value in the methods table must be ... giving a complete description of and Java Virtual Machine code for a method in the class or interface.*

Presumably, an entry in the methods table must include Java Virtual Machine code for a method if, and only if, that method is not flagged as ACC_ABSTRACT or ACC_NATIVE.

Furthermore, in the next paragraph it is stated that

> *Interfaces have only the single method <clinit>, ...*

Presumably, interfaces may declare class and instance methods as well, provided those methods are flagged as ACC_ABSTRACT.

p. 95 Concerning the class_index of a field (or method) reference, it is stated that

> *The class_index item of a CONSTANT_Fieldref_info ... structure must be a class type, not an interface type.*

However, this contradicts the following statement on page 147:

> *A constant pool entry tagged as CONSTANT_Fieldref represents a class or instance variable or a (constant) field of an interface.*

Presumably, a field reference may refer to a (constant) field declared in an interface.

p. 95 Restrictions on the constant pool entry referred to by a name_and_type_index are not fully defined; presumably, the descriptor associated with a field reference must be a field descriptor, and the descriptor associated with a method reference (or interface method reference) must be a method descriptor.

p. 104 Table 4.4 indicates that the flag ACC_STATIC can be used by any class or instance method; similarly, in the first paragraph on page 105, it is stated that

> *Class and instance methods may use any of the flags in Table 4.4.*

Presumably, an instance method is a method that is not flagged as being ACC_STATIC.

p. 105 Restrictions on combinations of method and class/interface access flags are not fully defined; presumably, a method may be flagged as ACC_ABSTRACT only if the class file in which it is declared represents an interface, or a class flagged as ACC_ABSTRACT.

p. 110 In the description of the Code attribute for a method it is stated that

There must be exactly one Code *attribute in each* method_info *structure.*

Apparantly, this could mean that an abstract method should have a Code attribute containing an empty code array. On page 111, however, it is stated that

> *The value of* code_length *must be greater than zero; the* code *array must not be empty.*

Presumably, there must be a Code attribute in the method_info structure for each method that is not flagged as ACC_ABSTRACT.

p. 112 The order of entries in the exception_table of a Code attribute is not described; presumably, the order of declaration for exception handlers is significant.

p. 117 Concerning the LocalVariableTable attribute for a method, it is stated that

> *The given local variable must have a value ... between* start_pc *and* start_pc+length *inclusive.*

However, this contradicts the statement following two lines below:

> *The value of* start_pc+length *must be either ... or the first index beyond the end of that* code *array.*

Presumably, a local variable cannot have a value at index start_pc+length, if start_pc+length = code_length.

p. 123 In the description of restrictions on use of the `getfield` and `putfield` instructions, it is explained that

> *The type of every class instance loaded from or stored into by a* getfield *or* putfield *instruction must be an instance of the class type or a subclass of the class type.*

In the type system of Java, a type cannot be an instance of any (class) type; furthermore, it is not clear which class the type of the instance is supposed to be an instance of.

Presumably, the class of an object loaded from or stored into by a `getfield` or `putfield` instruction must be the class declaring the accessed instance field, or a subclass thereof.

p. 143 Concerning verification of access permissions for a class, it is mentioned that a conflict might arise if

> *... a class that is originally declared* public *is changed to be* private *...*

Presumably, a class cannot be declared to be private; there is no such class modifier, and the ACC_PRIVATE flag cannot be used as class access flag.

p. 317 In the description of the `multianewarray` instruction, it is explained that

> *The components of the array of in the first dimension are initialized to subarrays of the type of the second dimension, and so on. The components of the first dimension of the array are initialized to the default initial value for the type of the components ...*

Presumably, it is the components of the *last* dimension of the array that are initialized to the default initial value for the type of the components, where the last dimension corresponds to the lowest level of sub-arrays being allocated by the `multianewarray` instruction.

References

[1] J. Gosling et. al. *The Java Language Specification.* Addison-Wesley, 1996.

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