A dozen instructions make Java bytecode[1]

Jacek Chrząszcz[2] Patryk Czarnik[3] Aleksy Schubert[4]

Institute of Informatics
University of Warsaw
ul. Banacha 2
02–097 Warsaw
Poland

Abstract

One of the biggest obstacles in the formalisation of the Java bytecode is that the language consists of 200 instructions. However, a rigorous handling of a programming language in the context of program verification and error detection requires a formalism which is compact in size. Therefore, the actual Java bytecode instruction set is never used in the context. Instead, the existing formalisations usually cover a 'representative' set of instructions. This paper describes how to reduce the number of instructions in a systematic and rigorous way into a manageable set of more general operations that cover the full functionality of the Java bytecode. The factorisation of the instruction set is based on the use of the runtime structures such as operand stack, heap etc. This is achieved by presentation of a formal semantics for the Java Virtual Machine.

Keywords: bytecode, semantics

1 Introduction

The transfer of programs from one party to the other raises the problem of security of its execution on the receiver's side. Therefore it is desirable to provide means to guarantee certain computational properties of the code in the form it travels from the developer to the consumer. Java bytecode language (JVML in short) is one of the most popular formats for a code that travels in the Internet and the security of its execution has already caused practical problems (see [5,8]) which go beyond the abilities to control the execution by means of Java sandboxing. One of the possible ways to overcome the problems is to provide a precise mathematical model for the language, then prove properties of the programs using the model and supply the travelling program with additional information that will make it possible to reconstruct the proof efficiently on the code consumer's side.

[1] This work was partly supported by Polish government grant N N206 493138.
[2] Email: `chrzaszcz@mimuw.edu.pl`
[3] Email: `czarnik@mimuw.edu.pl`
[4] Email: `alx@mimuw.edu.pl`

Several formal semantics were proposed for the JVML including the most notable ones: [1,7,11,13,14,15,16]. These formulations suffer from one of two problems — either they provide a formal semantics of (almost) all 200 bytecode instructions [5] or they choose a subset of the instructions that represents most of the interesting features. The drawback of the former option is that the formalisation in this case is very difficult to operate with as most of the proofs have to be done by induction on the structure of programs. Therefore the latter option is more often followed by researchers, but then the particular choice of instruction representatives is often not related to the actual instructions of the bytecode and is presented with very little discussion on the issue of the correspondence of the actual instructions to the ones in the model. The current paper provides the missing discussion and divides the instructions into groups that follow the same pattern of access to the JVM runtime structures (such as heap, operand stack etc.). For example, all load instructions are grouped together, all jumps, including the subroutine ones (`jsr` and `ret`), but also *`aload`, `getfield`, `checkcast` and `instanceof` form a single group as they all access the heap and (possibly) put something on the operand stack or raise an exception. In this way we obtain a factorisation of the whole set of the JVML instructions to 12 items. The actual lists of instructions are in the appendix.

We believe that it is crucial to come up with a formalisation that is based on a small number of instructions as then it is much easier to demonstrate the properties of the language itself — many proofs for such a language are done by induction on the structure of possible programs. If the number of instructions is limited then the number of cases to consider in such a proof is small. This is the main reason why ventures such as EML [10], where the number of semantical rules reaches several hundred, failed to develop metatheoretical properties, while such as Coq module system [2] succeeded with this regard. Moreover, it is a standard compiler design technique to establish a small language that makes easy design of optimisation techniques. Examples of such languages for Java and its bytecode include BAF, Jimple and Grimp [17] as well as BIR [6].

Moreover, our rigorous consideration gives the opportunity to present what are the instructions that really cover the whole spectrum of bytecode behaviours. We are aware that for certain properties of the JVML a slightly different set of instructions would be more convenient (e.g. the proofs for interval static analysis require access to the actual arithmetic operations and then it is desirable to consider them explicitly). However, one still has a path to reach to all the operations in JVML as their particular behaviour in our semantics is available through access to appropriate tables associated with our generalised operations. We hope that this solution is useful in all meta-proofs for JVML as it allows to build a common framework for many analyses which is important when a verification platform is to be built for real JVML programs.

Naturally, this paper does not provide the full semantics for the JVML as it is very complex. In fact, in a few places we make deliberate simplifications of the semantics in order to stay comprehensive in presentation.

This paper is an extended version of [3].

[5] The number is even greater when one considers wide instructions as separate.

2 Semantic domains and notation

We give here a small step semantics for the Java bytecode. The general form of a semantics step is:

$$P \vdash h, ts \to h', ts' \tag{2.1}$$

where P is a program, h, h' are heaps and ts, ts' are states of the threads. The semantic domains of these values are defined in the following way. First, we provide the description of programs: $\mathsf{Prog} = [\mathsf{Cnames} \rightharpoonup_{\mathit{fin}} \mathsf{CDesc}]$. Programs are partial functions with finite domain that associate class descriptions from CDesc with class names from Cnames. The class names are just appropriately defined identifiers, the class descriptions are defined as $\mathsf{CDesc} = [\mathsf{Mnames} \rightharpoonup_{\mathit{fin}} \mathsf{MDesc}]$ i.e. partial functions with finite domains that associate method descriptions MDesc with the method names. Again the method names are just appropriate identifiers while the method descriptions are somewhat more complicated and defined as

$$\mathsf{MDesc} = [\mathsf{PC} \rightharpoonup_{\mathit{fin}} \mathsf{Instr}] \times \mathsf{ExTable}$$
$$\mathsf{ExTable} = [\mathsf{PC} \times \mathsf{Cnames} \rightharpoonup_{\mathit{fin}} \mathsf{PC}]$$

where Instr is the set of JVM instructions and $\mathsf{ExTable}$ is an exception table for the method. The intent is that a function in $[\mathsf{PC} \rightharpoonup_{\mathit{fin}} \mathsf{Instr}]$ provides a mapping from instruction labels to the instructions under the labels. The $\mathsf{ExTable}$ returns the handler address for a given exception origin address and class.

The set of heaps is defined to be the set of

$$\mathsf{Heap} = [\mathsf{Loc} \times \mathsf{ThreadId} \rightharpoonup_{\mathit{fin}} (\mathsf{Cnames} \times \mathsf{Monitor} \times [\mathsf{Fnames} \rightharpoonup_{\mathit{fin}} \mathsf{Val}])]$$

where Loc is the set of locations (e.g. natural numbers or pointers in the current architecture) with a distinguished location `null` (the set $\mathsf{Loc}\backslash\{\texttt{null}\}$ will be denoted by $\mathsf{Loc}^{\bullet}$), $\mathsf{ThreadId}$ is the set of the thread identifiers (e.g. natural numbers), $\mathsf{Monitor}$ is the set of monitors which will control the lock counter for the given object, this is defined precisely later. Fnames is the set of field names and Val is the set of expected field values i.e. $\mathsf{Val} = \mathsf{int} \uplus \mathsf{long} \uplus \cdots \uplus \mathsf{Loc}$. The $\mathsf{ThreadId}$ is an argument of the heap, as each thread has its own view of the heap state. The exact way the different views are synchronised is described by the Java Memory Model [9, Section 17].

The set of *thread states* is defined as the set of all finite sets of thread descriptions, $P_{\mathit{fin}}(\mathsf{Thread}) \times \mathsf{History}$ combined with a state information $\mathsf{History}$ which contains an information needed for the thread scheduler to deterministically select a thread to execute. A thread description is

$$\mathsf{Thread} = \mathsf{ThreadId} \times \mathsf{ThreadStatus} \times \mathsf{EvalState} \times \mathsf{FrameStack}$$

where $\mathsf{ThreadStatus}$ represents the current status of the thread i.e. sleeping, blocked, running, terminated etc. At last, the $\mathsf{FrameStack} = \mathsf{MethodFrame}^{*}$ contains a sequence of the method frames of the form

$$\mathsf{MethodFrame} = \mathsf{Cnames} \times \mathsf{Mnames} \times \mathsf{LVals} \times \mathsf{OpStack} \times \mathsf{PC}$$

where LVals is the local variable table defined as $[\mathit{Vars} \rightharpoonup_{\mathit{fin}} \mathsf{Type} \times \mathsf{Val}]$ with the set of local variable indices $\mathit{Vars} = \mathbb{N}$, Type being the type of the value in the given entry and Val the value contained in the local variable table; $\mathsf{OpStack}$ is the operand stack defined as $(\mathsf{StackKind} \times (\mathsf{Val} \uplus \mathsf{PC}))^*$, where $\mathsf{StackKind}$ represents the type of the value in the current cell of the stack, note that we have to add PC type to make sure we can put labels of bytecode instructions used by subroutine commands; the same set PC is used as the final compound of $\mathsf{MethodFrame}$ and the value points to the currently executed bytecode instruction; $\mathsf{EvalState}$ is a set that represents the information on which exception has been thrown. We may assume that $\mathsf{EvalState} = \mathsf{Loc}$. The special location `null` is used to mark the situation that no exception has been thrown. We also assume that certain exceptions, such as `NullPointerException`, `ClassCastException` etc. are preallocated on the heap. This greatly simplifies the semantics as otherwise a number of semantic rules would be needed to allocate the exception on the stack and call its constructor before actially throwing it. And since the simplification does not concern used defined exceptions we decided not to complicate the semantics. [6]

It is worth mentioning that the semantics we provide here is in the so called *defensive* style i.e. we provide the type identification along with the operand stack and local variables table entries to check if the values stored there have correct type.

We can now define the set of monitors $\mathsf{Monitor}$ to be the product $\mathsf{ThreadId} \times \mathbb{N}$. A pair from the set represents the identifier of the thread that holds the lock and the number of the times the thread entered the monitor. We assume that the set $\mathsf{ThreadId}$ contains a distinguished constant `none` which is used to represent the situation when no thread holds the monitor.

A natural operation on the operand stack o is pushing an element e. It is written as $e \cdot o$. The examining the top of the stack is done by pattern matching and $o = e \cdot o'$ means that the stack o contains e at the top followed by the rest in o'.

The data structures which describe the state of the virtual machine are complicated. Therefore we need further notation to retrieve the information from them. First, we have to introduce the scheduler which chooses the particular thread to be executed: $\circledast : P_{\mathit{fin}}(\mathsf{Thread}) \times \mathsf{History} \to \mathsf{Thread}$. We do not provide a particular definition for $\mathsf{History}$ as this is implementation dependent. We assume only that the scheduler returns any element from its first argument. To make the notation more succint we write $\circledast_{ts}$ to denote $\circledast(ts)$. The components of the current thread are denoted as $\circledast_{ts} = \langle \mathrm{tid}_{ts}, \mathrm{tstatus}_{ts}, \mathrm{est}_{ts}, \mathrm{tfs}_{ts} \rangle$. As tfs_{ts} is also a composite value, we introduce further notation

$$\mathrm{tfs}_{ts} = \langle \mathrm{cnm}_{ts}, \mathrm{mnm}_{ts}, \mathrm{lv}_{ts}, \mathrm{ostck}_{ts}, \mathrm{pc}_{ts} \rangle \cdot \mathrm{tfs}_{ts}^{\mathsf{tail}} \tag{2.2}$$

where cnm_{ts} is the class name and mnm_{ts} is the method name of the currently executed method, lv_{ts} is the local variables table for the current method, ostck_{ts} is the current operand stack, pc_{ts} is the label of the currently executed instruction. The value $\mathrm{tfs}_{ts}^{\mathsf{tail}}$ denotes the (possibly empty) sequence of remaining method frames on the frame stack.

[6] In the Bicolano [13] JVM semantics the space on the heap is allocated but the constructor is not called.

2.1 Modification and lookup notation

We frequently modify slightly a given thread state to obtain a new one. The modification is described using the notation *changed_item*[*replaced_part* $\leftarrow$ *new_part*]. These can be defined precisely as the construction of a new value where all components but *replaced_part* are unchanged and the latter is replaced by *new_part*. For example $\mathrm{tfs}[\mathrm{lv} \leftarrow \mathrm{lv}']$ is a thread state tfs modified so that its local variable table lv_{ts} in the topmost method frame is replaced with a new table lv'.

The lookup of a particular instruction is done using the notation $P@\mathrm{pc.mnm.cnm}$ where $P \in \mathsf{Prog}$, $\mathrm{pc} \in \mathsf{PC}$, $\mathrm{mnm} \in \mathsf{Mnames}$, and $\mathrm{cnm} \in \mathsf{Cnames}$. This operation extracts from the program P the class declaration cnm and then it uses the Java method lookup scheme to retrieve the method of the name mnm (we assume the method name is such that it takes into account the signature of the method and therefore uniquely determines the method in the class). Then pc indicates which bytecode instruction from the code of the method should be retrieved.

Similarly, $P@\mathrm{etable.mnm.cnm}$ denotes the exception table for the method of the name mnm in the class cnm in P.

For $h \in \mathsf{Heap}$, $s \in \mathsf{Loc}$, and $i \in \mathsf{ThreadId}$ we write $h(s,i)$ to denote the value at the location s visible in the heap h from the thread i. In most cases i is clear from the context so we omit it and write $h(s)$. As $h(s)$ is a compound value, we define

$$h(s)@\mathrm{cnm} = \pi_1(h(s)) \qquad h(s)@\mathrm{monitor} = \pi_2(h(s)) \qquad h(s)@\mathrm{obj} = \pi_3(h(s))$$

$$h(s)@\mathrm{tid} = \pi_1(\pi_2(h(s))) \qquad h(s)@\mathrm{lcount} = \pi_2(\pi_2(h(s)))$$

In case $s = \texttt{null}$ or $s \notin \mathrm{dom}(h)$, the notations above have the value $\perp$.

2.2 Auxiliary definitions

Throughout the following semantics description we use many minor notations. This section collects the description of their meaning.

The names such as `int` are used here in two meanings, as a name for the set of elements in the Java type of native integers and as a syntactical identifier which is used to refer to the set. The 64-bit values divide into two halves. The notation $\texttt{long}(m_1, m_2)$ (resp. $\texttt{double}(m_1, m_2)$) means the 64-bit value of type `long` (resp. `double`) constructed from two 32-bit words m_1 and m_2. The type of a half with no distinction to which half and for which type (`long` or `double`) for a 64-bit value is denoted as `half`.

The Java Virtual Machine handles the 64-bit types in a special way. Therefore, the Java computational kinds are divided according to [12, Section 3.11.1] in two categories: $\mathsf{Cat1} = \{\texttt{int}, \texttt{float}, \texttt{ref}, \texttt{returnAddr}\}$ for 32-bit types and $\mathsf{Cat2} = \{\texttt{long}, \texttt{double}\}$ for 64-bit types. We will also use the notation $\mathsf{Cat1}^\bullet$ to denote $\mathsf{Cat1} \setminus \{\texttt{returnAddr}\}$.

As soon as a current thread is chosen we can conclusively determine the currently executed method. This method is denoted $\mathsf{cmthd} \in \mathsf{MDesc}$. We also use a function $\mathsf{next} : \mathsf{MDesc} \times \mathsf{PC} \to \mathsf{PC}$ to obtain the label of the next instruction in the method using the order of the instruction occurrence there.

2.3 Additional remarks

The semantics we give below is in fact more in the flavour of the interleaving semantics than the actual Java Memory Model one. However, we provide here a way to handle the Java Memory Model as our heap is defined so that it can give a different view of the memory to each thread. Other features of the semantics such as class loading, class initialisation, finalisation, native and synchronized methods etc. are not handled as well. However, slight changes of the definitions above can give the rules below the meaning which can take them into account. Adding reflection would be more problematic as it would require us to change the form of semantic steps.

3 Semantics of instructions

The semantic rules present the evolution of runtime structures caused by the execution of instructions. Most of the rules are directly governed by the current instruction of the current method, but those dealing with exceptions are not.

In the course of the semantic transition the scheduler $\circledast$ chooses a particular thread in ts to be executed. The notations we introduced in Section 2.1 all rely on the assumption that a thread is fixed. Therefore, we fix a single choice made by $\circledast$ throughout each particular rule. However, the choice may change for different steps of our semantics. We also assume that the state of the heap can change after each rule so that the visibility of its content gets partially synchronised among threads. If we do full synchronisation with every step we obtain the interleaving semantics.

3.1 Instruction load

This instruction generalizes all JVM instructions that read local variables and push the value to the operand stack. Its parameters describe the type and source of the value to be written to the stack, the general form of the instruction is $\mathit{load}(k, n)$ where $k \in \mathsf{Cat1}^{\bullet} \cup \mathsf{Cat2}$ is a kind, and n is a local variable index.

In the simplest case, when k is a 32-bit kind, $k \in \mathsf{Cat1}^{\bullet}$, the instruction reads a value from the local variable pointed by the index n and puts the value on the top of the operand stack. It is required that the value is of kind k.

$$\frac{\begin{array}{c}\mathrm{lv}_{ts}(n) = (k, m) \quad \mathrm{ostck}' = (k, m) \cdot \mathrm{ostck}_{ts} \quad \mathrm{pc}' = \mathsf{next}(\mathsf{cmthd}, \mathrm{pc}_{ts}) \\ P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = \mathit{load}(k, n) \quad k \in \mathsf{Cat1}^{\bullet} \quad \mathrm{est}_{ts} = \mathtt{null}\end{array}}{P \vdash h, ts \to h, ts[\mathrm{ostck} \leftarrow \mathrm{ostck}'][\mathrm{pc} \leftarrow \mathrm{pc}']} \ \mathit{ncat1\text{-}load} \tag{3.1}$$

If k denotes a category-2 kind (`long` or `double`), the value to push on the stack is obtained from the values of two variables, indexed by n and $n + 1$. This is because category-2 values occupy two subsequent cells in the local variables array. We provide an artificial kind `half` for the second variable in such a pair of variables. Following the JVM description [12, Section 3.6.2] we use a single operand stack element for a category-2 value.

$$\frac{\begin{array}{c}\mathrm{lv}_{ts}(n) = (k, m_1) \quad \mathrm{lv}_{ts}(n+1) = (\texttt{half}, m_2) \\ \mathrm{ostck}' = (k, k(m_1, m_2)) \cdot \mathrm{ostck}_{ts} \quad \mathrm{pc}' = \mathsf{next}(\mathsf{cmthd}, \mathrm{pc}_{ts}) \\ P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = \mathit{load}(k, n) \quad k \in \mathsf{Cat2} \quad \mathrm{est}_{ts} = \texttt{null}\end{array}}{P \vdash h, ts \to h, ts[\mathrm{ostck} \leftarrow \mathrm{ostck}'][\mathrm{pc} \leftarrow \mathrm{pc}']}\ \textit{ncat2-load} \tag{3.2}$$

3.2 Instruction store

This instruction generalizes all JVM instructions that pop a value from the operand stack and put it in the local variable table. Its arguments are the kind and destination of the popped value, the general form of the instruction is $\mathit{store}(k, n)$ where $k \in \mathsf{Cat1}^\bullet \cup \mathsf{Cat2}$ is a kind and n is a local variable index.

In case of a category-1 kind, the *store* instruction pops the topmost value from the operand stack and stores it in a local variable indexed by n.

$$\frac{\begin{array}{c}\mathrm{lv}' = \mathrm{lv}_{ts}[n \leftarrow (k, m)] \\ \mathrm{ostck}_{ts} = (k, m) \cdot \mathrm{ostck}' \quad \mathrm{pc}' = \mathsf{next}(\mathsf{cmthd}, \mathrm{pc}_{ts}) \\ P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = \mathit{store}(k, n) \quad k \in \mathsf{Cat1}^\bullet \quad \mathrm{est}_{ts} = \texttt{null}\end{array}}{P \vdash h, ts \to h, ts[\mathrm{ostck} \leftarrow \mathrm{ostck}'][\mathrm{pc} \leftarrow \mathrm{pc}'][\mathrm{lv} \leftarrow \mathrm{lv}']}\ \textit{ncat1-store} \tag{3.3}$$

If $k \in \mathsf{Cat2}$, two subsequent variables, n and $n+1$, are modified. It is required that the first variable is of kind k, and the second one is of kind `half`.

$$\frac{\begin{array}{c}\mathrm{lv}' = \mathrm{lv}_{ts}[n \leftarrow (k, m_1)][n+1 \leftarrow (\texttt{half}, m_2)] \\ \mathrm{ostck}' = (k, k(m_1, m_2)) \cdot \mathrm{ostck}_{ts} \quad \mathrm{pc}' = \mathsf{next}(\mathsf{cmthd}, \mathrm{pc}_{ts}) \\ P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = \mathit{store}(k, n) \quad k \in \mathsf{Cat2} \quad \mathrm{est}_{ts} = \texttt{null}\end{array}}{P \vdash h, ts \to h, ts[\mathrm{ostck} \leftarrow \mathrm{ostck}'][\mathrm{pc} \leftarrow \mathrm{pc}']}\ \textit{ncat2-store} \tag{3.4}$$

3.3 Instruction stackop

Instruction $\mathit{stackop}(\mathsf{op})$ generalizes all JVM instructions that use only the operand stack. It should be noted, that all such instructions operate on a fixed number of top elements, while the bottom part of the stack is neither read nor modified.

The parameter op denotes the stack operation to perform. The meaning of op is obtained through $\mathsf{kinds}_{\mathit{stackop}}(\mathsf{op})$, which is a set of triples, each of them consisting of: a list of input kinds l, a function f, and a list of output kinds l'.

The list l defines the requirements of the operation with respect to the operand stack. The number of stack elements must not be less than the length of l, and for all i, the i-th element of the stack must be of kind l_i. This is denoted by $\mathsf{check}(s, l)$.

The function $f : \mathsf{OpStack} \to \mathsf{OpStack}$ is the actual stack operation. $|l|$ elements are popped from the stack and become the input of f, then the result of f is pushed on the stack; l' describes guaranteed kinds of the result of f. In a sense $f : l \to l'$.

$$\frac{\begin{array}{c}(l, f, l') \in \mathsf{kinds}_{\mathit{stackop}}(\mathsf{op}) \qquad \mathrm{ostck}_{ts} = s \cdot r \\ \mathsf{check}(s, l) \quad \mathrm{ostck}' = f(s) \cdot r \quad \mathrm{pc}' = \mathsf{next}(\mathsf{cmthd}, \mathrm{pc}_{ts}) \\ P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = \mathit{stackop}(\mathsf{op}) \qquad \mathrm{est}_{ts} = \mathtt{null}\end{array}}{P \vdash h, ts \to h, ts[\mathrm{ostck} \leftarrow \mathrm{ostck}'][\mathrm{pc} \leftarrow \mathrm{pc}']}\ \textit{n-stackop} \tag{3.5}$$

For example, the JVM instruction `iadd` is mapped to *stackop*(iadd), and

$$\mathsf{kinds}_{\mathit{stackop}}(\mathsf{iadd}) = \{([\mathtt{int}, \mathtt{int}], f_{\mathsf{iadd}}, [\mathtt{int}])\}$$

where f_{iadd} performs addition of two 32-bit integers.

Polymorphic instructions, such as `swap` or `dup`, have more than one item in $\mathsf{kinds}_{\mathit{stackop}}$, for instance $\mathsf{kinds}_{\mathit{stackop}}(\mathsf{dup2})$ is equal to

$$\{([k_1, k_2], f_{\mathsf{dup2}}, [k_1, k_2, k_1, k_2])\}_{k_1, k_2 \in \mathsf{Cat1}} \ \cup \ \{([k], f_{\mathsf{dup}}, [k, k])\}_{k \in \mathsf{Cat2}}$$

3.4 Instruction cond

This instruction generalizes all JVM instructions that may affect the program control flow inside the current method, but do not modify the method frame stack, that is all unconditional and conditional jumps including `tableswitch`, `lookupswitch`, `jsr` and `ret`. The instruction reads and modifies the operand stack and the program counter (PC). The general form of the instruction is $\mathit{cond}(\mathsf{op}, d)$ where op identifies the actual operation on runtime structures and $d \in D_{\mathit{cond}}$, $D_{\mathit{cond}} = [\mathbb{N} \rightharpoonup_{\mathit{fin}} \mathsf{PC}]$ represents the static arguments of the instruction, which consist of an indexed table of addresses. The form and role of $\mathsf{kinds}_{\mathit{cond}}(\mathsf{op})$ is analogous to the role of $\mathsf{kinds}_{\mathit{stackop}}$. The difference here is the type of $f : D_{\mathit{cond}} \times \mathsf{OpStack} \times \mathsf{PC} \to \mathsf{OpStack} \times \mathsf{PC}$.

Arguments of f are the table of offsets, the relevant part of the operand stack, and the next PC. The function f returns the new value of the relevant part of the operand stack and the new value of PC. Only one JVM jump instruction, `jsr`, does put some value onto the operand stack: the current PC; `ret` is the only instruction that pops the new value of PC from the operand stack.

$$\frac{\begin{array}{c}(l, f, l') = \mathsf{kinds}_{\mathit{cond}}(\mathsf{op}) \qquad \mathrm{ostck}_{ts} = s \cdot r \\ \mathsf{check}(s, l) \quad (s', \mathrm{pc}') = f(d, s, \mathsf{next}(\mathsf{cmthd}, \mathrm{pc}_{ts})) \quad \mathrm{ostck}' = s' \cdot r \\ P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = \mathit{cond}(\mathsf{op}, d) \qquad \mathrm{est}_{ts} = \mathtt{null}\end{array}}{P \vdash h, ts \to h, ts[\mathrm{ostck} \leftarrow \mathrm{ostck}'][\mathrm{pc} \leftarrow \mathrm{pc}']}\ \textit{n-cond} \tag{3.6}$$

For example, the JVM instruction `ifeq(`o`)`, performing a jump if the value on the top of the stack is the integer 0, is mapped to $\mathit{cond}(\mathsf{ifeq}, [0 \mapsto \mathrm{pc} + o])$, and $\mathsf{kinds}_{\mathit{cond}}(\mathsf{ifeq}) = ([\mathtt{int}], f_{\mathsf{ifeq}}, [\,])$ with $f_{\mathsf{ifeq}}(g, s, \mathrm{pc})$ returning $([\,], g(0))$ if $s = [(\mathtt{int}, 0)]$ and $([\,], \mathrm{pc})$ otherwise. For `lookupswitch`, g is a function that maps key values to the corresponding addresses.

3.5 Instruction iinc

The opcode `iinc` is the only JVM instruction that uses solely the local variables array. The corresponding instruction in our formalisation is $\mathit{iinc}(n, c)$, where n is a local variable index and c is an integer value.

If the local variable n is of kind `int`, its value is increased by c, according to the Java `int` arithmetic.

$$\frac{\begin{array}{c}\mathrm{lv}_{ts}(n) = (\mathtt{int}, m) \quad \mathrm{lv}' = \mathrm{lv}_{ts}[n \leftarrow (\mathtt{int}, m +_{\mathtt{int}} c)] \\ \mathrm{pc}' = \mathsf{next}(\mathsf{cmthd}, \mathrm{pc}_{ts}) \quad P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = \mathit{iinc}(n, c) \quad \mathrm{est}_{ts} = \mathtt{null}\end{array}}{P \vdash h, ts \to h, ts[\mathrm{lv} \leftarrow \mathrm{lv}'][\mathrm{pc} \leftarrow \mathrm{pc}']}\ \textit{n-iinc} \tag{3.7}$$

3.6 Instruction get

This instruction reads the heap and modifies the operand stack. The general form of the instruction is $\mathit{get}(\mathsf{op}, d)$, where op is the operator and d contains an optional static argument—a qualified field name.

As for the previous rules, $\mathsf{kinds}_{get}(\mathsf{op}, d)$ provides expected kinds of arguments on the stack, list of kinds of values to be put on the stack, and the function f of type $D_{get} \times \mathsf{OpStack} \times \mathsf{Heap} \to \mathsf{OpStack} \uplus \mathsf{Loc}^{\bullet}$. The function f attempts to read the indicated object field or array cell from the heap. If it exists, f returns the modified part of the stack, which is the value from the heap.

$$\frac{\begin{array}{c}(l, f, l') = \mathsf{kinds}_{get}(\mathsf{op}, d) \quad \mathrm{ostck}_{ts} = s \cdot r \quad \mathsf{check}(s, l) \\ s' = f(d, s, h) \quad s' \in \mathsf{OpStack} \quad \mathrm{ostck}' = s' \cdot r \\ \mathrm{pc}' = \mathsf{next}(\mathsf{cmthd}, \mathrm{pc}_{ts}) \quad P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = \mathit{get}(\mathsf{op}, d) \quad \mathrm{est}_{ts} = \mathtt{null}\end{array}}{P \vdash h, ts \to h, ts[\mathrm{ostck} \leftarrow \mathrm{ostck}'][\mathrm{pc} \leftarrow \mathrm{pc}']}\ \textit{n-get} \tag{3.8}$$

If it is impossible to obtain the requested value and an exception must be thrown (e.g. `NullPointerException`), f returns the location e of the exception in the heap and the resulting evaluation state is the exceptional state.

$$\frac{\begin{array}{c}(l, f, l') = \mathsf{kinds}_{get}(\mathsf{op}, d) \quad \mathrm{ostck}_{ts} = s \cdot r \quad \mathsf{check}(s, l) \\ e = f(d, s, h) \quad e \in \mathsf{Loc}^{\bullet} \quad P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = \mathit{get}(\mathsf{op}, d) \quad \mathrm{est}_{ts} = \mathtt{null}\end{array}}{P \vdash h, ts \to h, ts[\mathrm{est} \leftarrow e]}\ \textit{exn-get} \tag{3.9}$$

3.7 Instruction put

This instruction reads and modifies the operand stack and the heap without creating new locations. The general form of the instruction is $\mathit{put}(\mathsf{op}, d)$, where op is the operator and d contains an optional static argument—a qualified field name.

The role of $\mathsf{kinds}_{put}(\mathsf{op}, d)$ is similar to previous kinds with the function f of type $D_{put} \times \mathsf{OpStack} \times \mathsf{Heap} \to \mathsf{Heap} \uplus \mathsf{Loc}^{\bullet}$. The function f attempts to modify the indicated field or array cell in the heap. If the indicated item exists and may be changed, f returns the modified heap.

Note that the value written by *put* does not have to be accessible by other threads immediately. In fact, any part of heap may be synchronized with the thread cache at any point of program execution, with Java Memory Model constraints preserved. In particular, the two halfs of a category-2 value may be synchronized independently.

$$\frac{\begin{array}{c}(l,f,l') \in \mathsf{kinds}(\mathsf{op},d) \quad \mathrm{ostck}_{ts} = s \cdot r \quad \mathsf{check}(s,l) \quad \mathrm{ostck}' = r \\ h' = f(d,s,h) \quad h' \in \mathsf{Heap} \quad \mathrm{pc}' = \mathsf{next}(\mathsf{cmthd}, \mathrm{pc}_{ts}) \\ P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = put(\mathsf{op},d) \qquad \mathrm{est}_{ts} = \mathtt{null}\end{array}}{P \vdash h, ts \to h', ts[\mathrm{ostck} \leftarrow \mathrm{ostck}'][\mathrm{pc} \leftarrow \mathrm{pc}']}\ \textit{n-put} \tag{3.10}$$

If the requested object does not exist, an exception is thrown.

$$\frac{\begin{array}{c}(l,f,l') \in \mathsf{kinds}(\mathsf{op},d) \quad \mathrm{ostck}_{ts} = s \cdot r \quad \mathsf{check}(s,l) \quad \mathrm{ostck}' = r \\ e = f(d,s,h) \quad e \in \mathsf{Loc}^{\bullet} \\ P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = put(\mathsf{op},d) \quad \mathrm{est}_{ts} = \mathtt{null}\end{array}}{P \vdash h, ts \to h, ts[\mathrm{est} \leftarrow e]}\ \textit{exn-put} \tag{3.11}$$

3.8 Instruction new

This instruction modifies the operand stack and the heap by creating a new location. The general form of the instruction is $new(\mathsf{op}, d)$, where op is the operator and d is a list of its arguments (integers and class names).

The precise meaning of the instruction is given by the function f, obtained from $\mathsf{kinds}_{new}(\mathsf{op}, d)$, together with expected kinds of arguments on the stack and the expected kinds of values to be stored on the operand stack, which is actually always one value of kind `ref`. The function f itself manipulates the heap, allocating the requested structure and returning the location of the allocated structure and the new heap in case of success, and the exception otherwise.

Note that this instruction and its rules are very similar to *put*. We preferred to keep the two separated as *new* adds new locations to the heap while *put* only modifies existing ones.

$$\frac{\begin{array}{c}(l,f,l') = \mathsf{kinds}_{new}(\mathsf{op},d) \quad \mathrm{ostck}_{ts} = s \cdot r \quad \mathsf{check}(s,l) \quad \mathrm{ostck}' = s' \cdot r \\ (s',h') = f(d,s,h) \quad s' \in \mathsf{OpStack} \quad h' \in \mathsf{Heap} \\ \mathrm{pc}' = \mathsf{next}(\mathsf{cmthd}, \mathrm{pc}_{ts}) \quad P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = new(\mathsf{op},d) \quad \mathrm{est}_{ts} = \mathtt{null}\end{array}}{P \vdash h, ts \to h', ts[\mathrm{ostck} \leftarrow \mathrm{ostck}'][\mathrm{pc} \leftarrow \mathrm{pc}']}\ \textit{n-new} \tag{3.12}$$

$$\frac{\begin{array}{c}(l, f, l') = \mathsf{kinds}_{new}(\mathsf{op}, d) \\ \mathrm{ostck}_{ts} = s \cdot r \quad \mathsf{check}(s, l) \quad e = f(d, s, h) \quad e \in \mathsf{Loc}^{\bullet} \\ P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = \mathit{new}(\mathsf{op}, d) \qquad \mathrm{est}_{ts} = \mathtt{null}\end{array}}{P \vdash h, ts \to h, ts[\mathrm{est} \leftarrow e]}\ \mathit{exn\text{-}new} \tag{3.13}$$

3.9 Instruction monitor

This instruction can modify the state of threads by trying to acquire or release a monitor. The operation itself is done by modifying an object on the heap. The *monitor* instruction expects one location on the operand stack: the object with which the monitor in question is associated. The general form of the instruction is $\mathit{monitor}(\mathsf{op})$, where op is either *enter* or *exit*.

Both variants of the instruction are handled by the same two rules — one for correct operation, one for raising an exception. The rules are governed by a partial function $f : \mathsf{ThreadId} \times \mathsf{Loc} \times \mathsf{ThreadId} \times \mathbb{N} \rightharpoonup \mathsf{ThreadId} \times \mathbb{N} \cup \mathsf{Loc}$ obtained from $\mathsf{kinds}_{monitor}(\mathsf{op})$. If $\mathsf{op} = \mathit{enter}$, $f(\mathrm{tid}', s, \mathrm{tid}, c)$ is defined only if $s = \mathtt{null}$ or $\mathrm{tid} = \mathtt{none}$ or $\mathrm{tid} = \mathrm{tid}'$. In the first case f returns a `NullPointerException`, in the second $(\mathrm{tid}', 1)$, and in the third $(\mathrm{tid}', c + 1)$. Since f is not defined when $s \neq \mathtt{null}$ and $\mathrm{tid}' \neq \mathrm{tid} \neq \mathtt{none}$, i.e. the monitor is owned by a different thread, the rule cannot be fired until the monitor is released.

If $\mathsf{op} = \mathit{exit}$, f returns the exception `IllegalMonitorStateException` if $\mathrm{tid} \neq \mathrm{tid}'$ and otherwise either `NullPointerException` or $(\mathtt{none}, 0)$ or $(\mathrm{tid}, c - 1)$ depending on the values of s and c.

For the lack of space we did not formalize other synchronization operations related to synchronized methods. Note however, that it is quite easy to syntactically transform a synchronized method into one having *monitor*(*enter*) at the beginning and *monitor*(*exit*) at every exit point.

$$\frac{\begin{array}{c}f = \mathsf{kinds}_{monitor}(\mathsf{op}) \quad \mathrm{ostck}_{ts} = s \cdot r \quad s \in \mathsf{Loc} \\ (\mathrm{tid}', \mathrm{lcount}') = f(\mathrm{tid}_{ts}, s, h(s)@\mathrm{tid}, h(s)@\mathrm{lcount}) \\ \mathrm{tid}' \in \mathsf{ThreadId} \quad \mathrm{lcount}' \in \mathbb{N} \quad \mathrm{pc}' = \mathsf{next}(\mathsf{cmthd}, \mathrm{pc}_{ts}) \quad \mathrm{ostck}' = r \\ h' = h[s \leftarrow h(s)[\mathrm{tid} \leftarrow \mathrm{tid}'][\mathrm{lcount} \leftarrow \mathrm{lcount}']] \\ P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = \mathit{monitor}(\mathsf{op}) \qquad \mathrm{est}_{ts} = \mathtt{null}\end{array}}{P \vdash h, ts \to h', ts[\mathrm{pc} \leftarrow \mathrm{pc}'][\mathrm{ostck} \leftarrow \mathrm{ostck}']}\ \mathit{n\text{-}monitor} \tag{3.14}$$

$$\frac{\begin{array}{c} f = \mathsf{kinds}_{monitor}(\mathsf{op}) \quad \mathrm{ostck}_{ts} = s \cdot r \quad s \in \mathsf{Loc} \\ e = f(\mathrm{tid}_{ts}, s, h(s)@\mathrm{tid}, h(s)@\mathrm{lcount}) \quad e \in \mathsf{Loc} \\ P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = monitor(\mathsf{op}) \qquad \mathrm{est}_{ts} = \mathtt{null} \end{array}}{P \vdash h, ts \rightarrow h, ts[\mathrm{est} \leftarrow \mathrm{e}]} \; exn\text{-}monitor \tag{3.15}$$

3.10 Instruction invoke

This instruction modifies the operand stack, the method frame stack and reads the heap. The general format of the instruction is $invoke(mode, \mathrm{cnm}, \mathrm{mnm})$, where $mode$ is one of `interface`, `special`, `static` or `virtual`, and cnm and mnm are class and method name of the method that is supposed to be called.

The principal action of this instruction is to find the method code, prepare the new method frame and pass the execution to the new method instance. To do that the types l of expected values on the stack together with the expected types return by the method l' are read from $\mathsf{kinds}_{invoke}(mode, \mathrm{cnm}, \mathrm{mnm})$, which in turn reads them from the method signature. The list l' is of length at most 1. Next, the *dispatch* function is executed which checks that the method's flags are not contradictory to the invoke $mode$, that the access rights are preserved (for `private` and `protected` methods) and selects the type of dispatch by returning either the class cnm for static dispatch or the class of the first location of s in h for dynamic dispatch. The *dispatch* function can also return an exception.

The rest of the *n-invoke* rule is devoted to the preparation of the new method frame: the function *initlv* places the arguments from the stack in the local variable table of the new frame after splitting values of type `long` and `double` and performing necessary floating-point value set conversions [12, Section 3.8.3]. Finally, the new method frame is put on the method frame stack with the empty initial operand stack and $\mathrm{pc} = 0$.

Synchronized methods are not handled here, but please see the remark at the end of Section 3.9.

$$\frac{\begin{array}{c} (l, l') = kindsof_{invoke}(mode, \mathrm{cnm}, \mathrm{mnm}) \quad \mathrm{ostck}_{ts} = s \cdot r \quad \mathsf{check}(s, l) \\ \mathrm{cnm}' = dispatch(mode, \mathrm{cnm}, \mathrm{mnm}, s, h) \quad \mathrm{tfs}' = \mathrm{tfs}_{ts}[\mathrm{ostck} \leftarrow r] \\ \mathrm{lv}' = initlv(lvlength(P@\mathrm{mnm}.\mathrm{cnm}), s) \\ \mathrm{tfs}'' = \langle \mathrm{cnm}', \mathrm{mnm}, \mathrm{lv}', [\,], 0 \rangle \cdot \mathrm{tfs}' \\ P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = invoke(mode, \mathrm{cnm}, \mathrm{mnm}) \quad \mathrm{est}_{ts} = \mathtt{null} \end{array}}{P \vdash h, ts \rightarrow h, ts[\mathrm{tfs} \leftarrow \mathrm{tfs}'']} \; n\text{-}invoke \tag{3.16}$$

$$\frac{\begin{array}{c}(l,l') = \mathit{kindsof}_{\mathit{invoke}}(\mathit{mode}, \mathrm{cnm}, \mathrm{mnm}) \quad \mathrm{ostck}_{ts} = s \cdot r \quad \mathsf{check}(s,l) \\ e = \mathit{dispatch}(\mathit{mode}, \mathrm{cnm}, \mathrm{mnm}, s, h) \quad e \in \mathsf{Loc} \\ P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = \mathit{invoke}(\mathit{mode}, \mathrm{cnm}, \mathrm{mnm}) \quad \mathrm{est}_{ts} = \mathtt{null}\end{array}}{P \vdash h, ts \to h, ts[\mathrm{est} \leftarrow e]}\ \mathit{exn\text{-}invoke} \tag{3.17}$$

3.11 Instruction return

This instruction returns from the current method. It reads the operand stack and modifies the method frame stack by removing the current frame and updating the previous frame: moving the pc to the next instructions (usually over an *invoke* instruction) and updating the operand stack by pushing the return value, after the floating-point value set conversion [12, Section 3.8.3]. The general form of the instruction is *return*(l) where l is a list of kinds of length at most 1.

Even though [12] does not specify this explicitly, we decided to add the rule *n-term-return*, to deal with the termination of the method corresponding to the last frame on the frame stack.

These rules do not handle releasing of monitor when exiting a synchronized method. This can be simulated, however, by putting a *monitor(exit)* instruction before every *return* statement. Please see also the discussion in Section 3.9.

$$\frac{\begin{array}{c}\mathrm{ostck}_{ts} = s \cdot r \quad \mathsf{check}(s,l) \\ \mathrm{tfs}_{ts} = f_1 \cdot \langle \mathrm{cnm}', \mathrm{mnm}', \mathrm{lv}', \mathrm{ostck}', \mathrm{pc}' \rangle \cdot \mathrm{tfs}^{\mathsf{tail}} \quad f_1 \in \mathsf{MethodFrame} \\ \mathrm{tfs}' = \langle \mathrm{cnm}', \mathrm{mnm}', \mathrm{lv}', \mathrm{vsc}(s) \cdot \mathrm{ostck}', \mathsf{next}(P@\mathrm{mnm}'.\mathrm{cnm}', \mathrm{pc}') \rangle \cdot \mathrm{tfs}^{\mathsf{tail}} \\ P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = \mathit{return}(l) \quad \mathrm{est}_{ts} = \mathtt{null}\end{array}}{P \vdash h, ts \to h, ts[\mathrm{tfs} \leftarrow \mathrm{tfs}']}\ \mathit{n\text{-}return} \tag{3.18}$$

$$\frac{\begin{array}{c}\mathrm{ostck}_{ts} = s \cdot r \quad \mathsf{check}(s,l) \quad \mathrm{tfs}_{ts} = [f] \quad f \in \mathsf{MethodFrame} \\ \mathrm{tfs}' = [] \qquad \mathrm{tstatus}' = \mathrm{TERMINATED} \\ P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = \mathit{return}(l) \quad \mathrm{est}_{ts} = \mathtt{null}\end{array}}{P \vdash h, ts \to h, ts[\mathrm{tfs} \leftarrow \mathrm{tfs}'][\mathrm{tstatus} \leftarrow \mathrm{tstatus}']}\ \mathit{n\text{-}term\text{-}return} \tag{3.19}$$

3.12 Instruction throw

This instruction takes no parameters, it reads and removes the location of the exception form the stack and changes the evaluation state of the current thread (the rule *ex-throw*). The way the exceptions are handled in our semantics is the following. The evaluation state (est) component of each thread says if the execution is in the normal state, when est = `null`, or in exception handling state otherwise.

Note that the switch to the latter state can be done not only by executing the *throw* instruction but also by throwing an exception (e.g. `NullPointerException`) by other semantic rules. If est $= e$ is a location of a valid exception, the remaining rules *ex-in-handle*, *ex-out-handle* or *ex-term-handle* can be fired, depending on the fact whether the exception is handled inside the current method or provokes its abrupt termination. In the latter case, the *ex-term-handle* rule handles the special case where the current method is the last on the method frame stack. This rule does not have a direct correspondence in [12], just like the rule *n-term-return*.

The feature which is not handled is the release of monitor when a synchronized method is abruptly terminated by an exception. Note however that this can be simulated by adding a catch-all exception handler which would execute the instruction *monitor(exit)* and then rethrow the exception. See also the discussion at the end of Section 3.9.

$$\frac{\begin{array}{c}\mathrm{ostck}_{ts} = e \cdot r \quad e \in \mathsf{Loc}^{\bullet} \\ P@\mathrm{pc}_{ts}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts} = \mathit{throw} \qquad \mathrm{est}_{ts} = \mathtt{null}\end{array}}{P \vdash h, ts \to h, ts[\mathrm{est} \leftarrow e]}\ \mathit{ex\text{-}throw} \tag{3.20}$$

$$\frac{\begin{array}{c}\mathrm{ostck}' = [e] \quad (\mathrm{pc}_{ts}, h(e)@\mathrm{cnm}) \in \mathrm{dom}(P@\mathrm{etable}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts}) \\ \mathrm{pc}' = P@\mathrm{etable}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts}(\mathrm{pc}_{ts}, h(e)@\mathrm{cnm}) \quad \mathrm{est}_{ts} = e \in \mathsf{Loc}^{\bullet}\end{array}}{P \vdash h, ts \to h, ts[\mathrm{ostck} \leftarrow \mathrm{ostck}'][\mathrm{pc} \leftarrow \mathrm{pc}'][\mathrm{est} \leftarrow \mathtt{null}]}\ \mathit{ex\text{-}in\text{-}handle} \tag{3.21}$$

$$\frac{\begin{array}{c}(\mathrm{pc}_{ts}, h(e)@\mathrm{cnm}) \notin \mathrm{dom}(P@\mathrm{etable}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts}) \\ \mathrm{tfs}_{ts} = f_1 \cdot f_2 \cdot \mathrm{tfs}_{ts}^{\mathsf{tail}} \quad f_1, f_2 \in \mathsf{MethodFrame} \quad \mathrm{est}_{ts} = e \in \mathsf{Loc}^{\bullet}\end{array}}{P \vdash h, ts \to h, ts[\mathrm{tfs} \leftarrow f_2 \cdot \mathrm{tfs}_{ts}^{\mathsf{tail}}]}\ \mathit{ex\text{-}out\text{-}handle} \tag{3.22}$$

$$\frac{\begin{array}{c}(\mathrm{pc}_{ts}, h(e)@\mathrm{cnm}) \notin \mathrm{dom}(P@\mathrm{etable}.\mathrm{mnm}_{ts}.\mathrm{cnm}_{ts}) \\ \mathrm{tfs}_{ts} = [f] \quad f \in \mathsf{MethodFrame} \quad \mathrm{est}_{ts} = e \in \mathsf{Loc}^{\bullet}\end{array}}{P \vdash h, ts \to h, ts[\mathrm{tfs} \leftarrow [\,]][\mathrm{tstatus} \leftarrow \mathrm{TERMINATED}]}\ \mathit{ex\text{-}term\text{-}handle} \tag{3.23}$$

3.13 Instructions without semantics

The functionality of a few instructions cannot be expressed by semantical transformation of the runtime structures as their meaning is not described in JVM specification [12]. These are `breakpoint`, `impdep1`, `impdep2`, and the instruction with the opcode 186.[7] Therefore, they are omitted from the paper. The opcode `wide` is taken into account along with the non-wide operations.

[7] JVM semantics says: 'For historical reasons, opcode value 186 is not used.'

4 Conclusions

We have presented a concise formalisation of JVML which turns out to be factorisable into 12 instruction mnemonics. This was possible because we separated generic operation of many instructions and tabularised particular behaviours of individual opcodes. In this way we rigorously reduced the overall complexity of the whole language without significantly sacrificing its features.

References

[1] Atkey, R., *CoqJVM: An executable specification of the Java Virtual Machine using dependent types*, in: M. Miculan, I. Scagnetto and F. Honsell, editors, *Types for Proofs and Programs, International Conference, TYPES 2007, Cividale des Friuli, Italy, May 2-5, 2007, Revised Selected Papers*, Lecture Notes in Computer Science **4941** (2008), pp. 18–32.

[2] Chrząszcz, J., *Modules in Coq are and will be correct*, in: S. Berardi, M. Coppo and F. Damiani, editors, *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, Lecture Notes in Computer Science **3085**, 2004, pp. 130–146.

[3] Chrząszcz, J., P. Czarnik and A. Schubert, *A dozen instructions make Java bytecode*, in: *Proceedings of Bytecode'2010*, 2010, to appear.

[4] Consortium, M., *Deliverable 3.1: Bytecode specification language and program logic* (2006), available online from http://mobius.inria.fr.

[5] Dean, D., E. Felten and D. Wallach, *Java security: From HotJava to Netscape and beyond*, Security and Privacy, IEEE Symposium on (1996), pp. 190–200.

[6] Demange, D., T. Jensen and D. Pichardie, *A provably correct stackless intermediate representation for Java bytecode*, Technical Report Research Report 7021, INRIA (2009).

[7] Freund, S. N., "Type systems for object-oriented intermediate languages," Ph.D. thesis, Stanford University (2000).

[8] Freund, S. N. and J. C. Mitchell, *The type system for object initialization in the Java bytecode language*, ACM Transaction on Programming Languages and Systems **21** (1999), pp. 1196–1250.

[9] Gosling, J., B. Joy, G. Steele and G. Bracha, "The Java Language Specification, third edition," The Java Series, Addison Wesley, 2005.

[10] Kahrs, S., D. Sannella and A. Tarlecki, *The definition of Extended ML: A gentle introduction*, Theoretical Computer Science **173** (1997), pp. 445–484.

[11] Klein, G. and T. Nipkow, *A machine-checked model for a Java-like language, virtual machine, and compiler*, ACM Transactions on Programming Languages and Systems **28** (2006), pp. 619–695.

[12] Lindholm, T. and F. Yellin, "The Java (TM) Virtual Machine Specification (Second Edition)," Prentice Hall, 1999.

[13] Pichardie, D., *Bicolano – Byte Code Language in Coq* (2006), http://mobius.inria.fr/bicolano. Summary appears in [4].

[14] Pusch, C., *Proving the soundness of a Java bytecode verifier specification in Isabelle/HOL*, in: R. Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, Lecture Notes in Computer Science **1579** (1999), pp. 89–103.

[15] Qian, Z., *A formal specification of Java Virtual Machine instructions for objects, methods and subrountines*, in: *Formal Syntax and Semantics of Java* (1999), pp. 271–312.

[16] Stärk, R. F., J. Schmid and E. Börger, "Java and the Java Virtual Machine: Definition, Verification, Validation," Springer, 2001.

[17] Vallée-Rai, R., P. Co, E. Gagnon, L. Hendren, P. Lam and V. Sundaresan, *Soot - a Java bytecode optimization framework*, in: *CASCON '99: Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research* (1999), p. 13.

A Factorisation of instructions

A.1 load

21 (0x15) iload
22 (0x16) lload
23 (0x17) fload
24 (0x18) dload
25 (0x19) aload
26 (0x1a) iload_0
27 (0x1b) iload_1
28 (0x1c) iload_2
29 (0x1d) iload_3
30 (0x1e) lload_0
31 (0x1f) lload_1
32 (0x20) lload_2
33 (0x21) lload_3
34 (0x22) fload_0
35 (0x23) fload_1
36 (0x24) fload_2
37 (0x25) fload_3
38 (0x26) dload_0
39 (0x27) dload_1
40 (0x28) dload_2
41 (0x29) dload_3
42 (0x2a) aload_0
43 (0x2b) aload_1
44 (0x2c) aload_2
45 (0x2d) aload_3

A.2 store

54 (0x36) istore
55 (0x37) lstore
56 (0x38) fstore
57 (0x39) dstore
58 (0x3a) astore
59 (0x3b) istore_0
60 (0x3c) istore_1
61 (0x3d) istore_2
62 (0x3e) istore_3
63 (0x3f) lstore_0
64 (0x40) lstore_1
65 (0x41) lstore_2
66 (0x42) lstore_3
67 (0x43) fstore_0
68 (0x44) fstore_1

69 (0x45) fstore_2
70 (0x46) fstore_3
71 (0x47) dstore_0
72 (0x48) dstore_1
73 (0x49) dstore_2
74 (0x4a) dstore_3
75 (0x4b) astore_0
76 (0x4c) astore_1
77 (0x4d) astore_2
78 (0x4e) astore_3

A.3 stackop

00 (0x00) nop
01 (0x01) aconst_null
02 (0x02) iconst_m1
03 (0x03) iconst_0
04 (0x04) iconst_1
05 (0x05) iconst_2
06 (0x06) iconst_3
07 (0x07) iconst_4
08 (0x08) iconst_5
09 (0x09) lconst_0
10 (0x0a) lconst_1
11 (0x0b) fconst_0
12 (0x0c) fconst_1
13 (0x0d) fconst_2
14 (0x0e) dconst_0
15 (0x0f) dconst_1
16 (0x10) bipush
17 (0x11) sipush
18 (0x12) ldc
19 (0x13) ldc_w
20 (0x14) ldc2_w
87 (0x57) pop
88 (0x58) pop2
089 (0x59) dup
090 (0x5a) dup_x1
091 (0x5b) dup_x2
092 (0x5c) dup2
093 (0x5d) dup2_x1
094 (0x5e) dup2_x2
095 (0x5f) swap
096 (0x60) iadd
097 (0x61) ladd
098 (0x62) fadd
099 (0x63) dadd

100 (0x64) isub
101 (0x65) lsub
102 (0x66) fsub
103 (0x67) dsub
104 (0x68) imul
105 (0x69) lmul
106 (0x6a) fmul
107 (0x6b) dmul
108 (0x6c) idiv
109 (0x6d) ldiv
110 (0x6e) fdiv
111 (0x6f) ddiv
112 (0x70) irem
113 (0x71) lrem
114 (0x72) frem
115 (0x73) drem
126 (0x7e) iand
127 (0x7f) land
128 (0x80) ior
129 (0x81) lor
130 (0x82) ixor
131 (0x83) lxor
148 (0x94) lcmp
149 (0x95) fcmpl
150 (0x96) fcmpg
151 (0x97) dcmpl
152 (0x98) dcmpg
116 (0x74) ineg
117 (0x75) lneg
118 (0x76) fneg
119 (0x77) dneg
120 (0x78) ishl
121 (0x79) lshl
122 (0x7a) ishr
123 (0x7b) lshr
124 (0x7c) iushr
125 (0x7d) lushr
133 (0x85) i2l
134 (0x86) i2f
135 (0x87) i2d
136 (0x88) l2i
137 (0x89) l2f
138 (0x8a) l2d
139 (0x8b) f2i
140 (0x8c) f2l
141 (0x8d) f2d

142 (0x8e) d2i
143 (0x8f) d2l
144 (0x90) d2f
145 (0x91) i2b
146 (0x92) i2c
147 (0x93) i2s

A.4 cond

153 (0x99) ifeq
154 (0x9a) ifne
155 (0x9b) iflt
156 (0x9c) ifge
157 (0x9d) ifgt
158 (0x9e) ifle
159 (0x9f) if_icmpeq
160 (0xa0) if_icmpne
161 (0xa1) if_icmplt
162 (0xa2) if_icmpge
163 (0xa3) if_icmpgt
164 (0xa4) if_icmple
165 (0xa5) if_acmpeq
166 (0xa6) if_acmpne
170 (0xaa) tableswitch
171 (0xab) lookupswitch
198 (0xc6) ifnull
199 (0xc7) ifnonnull
167 (0xa7) goto
200 (0xc8) goto_w
168 (0xa8) jsr
201 (0xc9) jsr_w
169 (0xa9) ret

A.5 iinc

132 (0x84) iinc

A.6 get

46 (0x2e) iaload
47 (0x2f) laload
48 (0x30) faload
49 (0x31) daload
50 (0x32) aaload
51 (0x33) baload
52 (0x34) caload
53 (0x35) saload
180 (0xb4) getfield

190 (0xbe) arraylength
192 (0xc0) checkcast
193 (0xc1) instanceof

A.7 put

79 (0x4f) iastore
80 (0x50) lastore
81 (0x51) fastore
82 (0x52) dastore
83 (0x53) aastore
84 (0x54) bastore
85 (0x55) castore
86 (0x56) sastore
179 (0xb3) putstatic
181 (0xb5) putfield

A.8 new

187 (0xbb) new
188 (0xbc) newarray
189 (0xbd) anewarray
197 (0xc5) multianewarray

A.9 monitor

194 (0xc2) monitorenter
195 (0xc3) monitorexit

A.10 invoke

182 (0xb6) invokevirtual
183 (0xb7) invokespecial
184 (0xb8) invokestatic
185 (0xb9) invokeinterface

A.11 return

172 (0xac) ireturn
173 (0xad) lreturn
174 (0xae) freturn
175 (0xaf) dreturn
176 (0xb0) areturn
177 (0xb1) return

A.12 throw

191 (0xbf) athrow

A.13 Instructions without semantics

196 (0xc4) wide
202 (0xca) breakpoint
254 (0xfe) impdep1
255 (0xff) impdep2