A Proposal for the JCVMLe Operational Semantics

Authors:	**Igor Siveroni and Chris Hankin**
Date:	**October 18, 2001**
Number:	**SECSAFE-ICSTM-001**
Version:	**2.2**
Classification:	**Internal**
Status:	**Work in progress**

1 Introduction

We present the operational semantics of the JCVMLe language, a language that models the Java Card Virtual machine Language. We use the *instruction set* and the *program structures* proposed in [4]. We define a small-step relation between program configurations, including rules for exception handling and subroutines. We also include the basic structures needed to model object ownership and the Java Card firewall.

New versions of this document should contain:

- A descrition of the interaction between the `JCRE` and the execution of applets.
- Definition of Java Card API methods that directly deal with applet and object initialization, applet selection, the Java Card firewall and object execution in general.
- A formal description of the static semantics of the machine.

One of the goals of the operational semantics is to provide an adequate framework for the formalisation of program (e.g. flow) analyses for the JCVM[2]. Further refinements to the semantics will be directed - among other things - towards this goal.

1.1 Notation

We follow the same notation conventions defined in [4]. In addition, we use the following:

Records. Some new domains are defined as records. Records are defined using the following notation:

$$Dom = (f_1 : Dom_1) \times ... \times (f_n : Dom_n)$$

Access to field f_i in $e \in Dom$ is written as $e.f_i$. Field update is written as $e.[f_i \mapsto f]$, where $f \in Dom_i$.

Conjunctions and Disjunctions. Conjunctions and disjunctions are grouped using parentheses and curly braces respectively. In particular, we write:

$$x = \begin{cases} v_1 & \text{if } cond_1 \\ ... & \\ v_n & \text{if } cond_n \end{cases}$$

instead of

$$\left\{\begin{array}{ll} (x = v_1) \wedge cond_1 & \vee \\ \ldots & \vee \\ (x = v_i) \wedge \neg cond_1 \wedge \ldots \wedge \neg cond_{i-1} \wedge cond_i & \vee \\ \ldots & \vee \\ (x = v_n) \wedge \neg cond_1 \wedge \ldots \wedge \neg cond_{n-1} \wedge cond_n & \end{array}\right\}$$

2 The JCVML$_e$ Syntax

2.1 The JCVML$_e$ Instruction Set

We divide the Java Card Bytecode Language Instruction Set into six groups[3][1]:

$$\begin{array}{rcl} I \in \mathit{Instruction} & ::= & \mathit{CoreInstruction} \mid \mathit{ObjectInstruction} \mid \mathit{MethodInstruction} \mid \mathit{ArrayInstruction} \\ & \mid & \mathit{ExceptionInstruction} \mid \mathit{SubroutineInstruction} \end{array}$$

We consider the language subset proposed in [4][2]:

$$\begin{array}{rcl} \mathit{CoreInstruction} & ::= & \texttt{nop} \mid \texttt{push}\ \mathit{OperandType}\ \mathit{Constant} \mid \texttt{pop}\ \mathit{integer} \\ & \mid & \texttt{dup}\ \mathit{NbWords}\ \mathit{NbWords} \mid \texttt{swap}\ \mathit{NbWords}\ \mathit{NbWords} \\ & \mid & \texttt{numop}\ \mathit{OperandType}\ \mathit{NumericOperator}\ \mathit{OperandType}_{opt} \\ & \mid & \texttt{load}\ \mathit{OperandType}\ \mathit{Index} \mid \texttt{store}\ \mathit{OperandType}\ \mathit{Index} \\ & \mid & \texttt{inc}\ \mathit{OperandType}\ \mathit{Index}\ \mathit{integer} \mid \texttt{goto}\ \mathit{Address} \\ & \mid & \texttt{if}\ \mathit{ComparisonOperator}\ \texttt{null}_{opt}\ \texttt{goto}\ \mathit{Address} \\ & \mid & \texttt{lookupswitch}\ \mathit{OperandType}\ (\mathit{integer}\ \texttt{=>}\ \mathit{Address})^*\ ,\ \texttt{default}\ \texttt{=>}\ \mathit{Address} \\ & \mid & \texttt{tableswitch}\ \mathit{OperandType}\ \mathit{integer}\ \texttt{=>}\ \mathit{Address}^*\ ,\ \texttt{default}\ \texttt{=>}\ \mathit{Address} \end{array}$$

$$\begin{array}{rcl} \mathit{ObjectInstruction} & ::= & \texttt{new}\ \mathit{ReferenceType} \\ & \mid & \texttt{checkcast}\ \mathit{ReferenceType} \mid \texttt{instanceof}\ \mathit{ReferenceType} \\ & \mid & \texttt{getstatic}\ \mathit{Field} \mid \texttt{putstatic}\ \mathit{Field} \\ & \mid & \texttt{getfield}\ \texttt{this}_{opt}\ \mathit{Field} \mid \texttt{putfield}\ \texttt{this}_{opt}\ \mathit{Field} \end{array}$$

$$\begin{array}{rcl} \mathit{MethodInstruction} & ::= & \texttt{invokedefinite}\ \mathit{Method} \mid \texttt{invokevirtual}\ \mathit{Method} \\ & \mid & \texttt{invokeinterface}\ \mathit{Method} \mid \texttt{return}\ \mathit{OperandType}_{opt} \end{array}$$

$$\begin{array}{rcl} \mathit{ArrayInstruction} & ::= & \texttt{arraylength} \mid \texttt{arrayload}\ \mathit{OperandType} \mid \texttt{arraystore}\ \mathit{OperandType} \\ \mathit{ExceptionInstruction} & ::= & \texttt{throw} \\ \mathit{SubroutineInstruction} & ::= & \texttt{jsr}\ \mathit{Address} \mid \texttt{ret}\ \mathit{Index} \end{array}$$

where

$$\begin{array}{rcl} \mathit{Constant} & ::= & \mathit{integer} \mid \texttt{null} \\ \mathit{NbWords} & ::= & \mathit{integer} \\ op \in \mathit{NumericOperator} & ::= & \mathit{UnaryNumericOperator} \mid \mathit{BinaryNumericOperator} \\ \mathit{UnaryNumericOperator} & ::= & \texttt{neg} \mid \texttt{to} \\ \mathit{BinaryNumericOperator} & ::= & \texttt{add} \mid \texttt{sub} \mid \texttt{mul} \mid \texttt{div} \mid \texttt{rem} \mid \texttt{cmp} \\ \mathit{ComparisonOperator} & ::= & \texttt{eq} \mid \texttt{ne} \mid \texttt{ge} \mid \texttt{gt} \mid \texttt{le} \mid \texttt{lt} \\ & \mid & \texttt{and} \mid \texttt{or} \mid \texttt{xor} \mid \texttt{shl} \mid \texttt{shr} \mid \texttt{ushr} \\ \mathit{Index} & ::= & \mathit{integer} \end{array}$$

[1] Integer and multi-word operations are considered by the current instruction set if the type `int` is used

[2] Non-terminals in *Instruction* correspond to instruction arguments. We use the syntax in the operational semantics rules in a similar way as pattern-matching is used to assign values to record fields. The interface - to extract the arguments - provided for instructions can be used instead.

An operand type,

$$t \in \mathit{OperandType} ::= \texttt{b} \mid \texttt{s} \mid \texttt{i} \mid \texttt{r},$$

indicates if the instruction is going to work with elements of type byte, short, integer or reference respectively. We reuse this notation in section 4 when we use run-time types. Similarly we reuse *ReferenceType*, defined in 2.2.

2.2 Types

JCVMLe values can have the following types[3]:

$$\begin{array}{rcl}
\tau \in \mathit{Type} & ::= & \mathit{ReferenceType} \mid \mathit{PrimitiveType} \\
\mathit{PrimitiveType} & ::= & \mathit{BooleanType} \mid \mathit{NumericType} \mid \mathit{ReturnAddressType} \\
\mathit{NumericType} & ::= & \mathit{IntegralType} \\
\mathit{IntegralType} & ::= & \texttt{byte} \mid \texttt{short} \mid \texttt{int} \\
\mathit{BooleanType} & ::= & \texttt{bool} \\
\mathit{ReturnAddressType} & ::= & \texttt{ret} \\
\tau_r \in \mathit{ReferenceType} & ::= & \mathit{ArrayType} \mid \mathit{ClassType} \mid \mathit{InterfaceType} \\
\mathit{ClassType} & ::= & \mathit{ClassName} \\
\mathit{InterfaceType} & ::= & \mathit{InterfaceName} \\
\mathit{ComponentType} & ::= & \mathit{BooleanType} \mid \mathit{NumericType} \mid \mathit{ClassType} \mid \mathit{InterfaceType} \\
\mathit{ArrayType} & ::= & (\texttt{array}\ \mathit{ComponentType})
\end{array}$$

A special type *MethodType* is defined:

$$\begin{array}{rcl}
\tau_m \in \mathit{MethodType} & = & (\mathit{parameterTypes} : [\mathit{Type}]) \quad \times \quad (\mathit{resultType} : \mathit{ResultType}) \\
\tau_r \in \mathit{ResultType} & = & \mathit{Type} \cup \mathit{VoidType}
\end{array}$$

but will usually write[4]:

$$\begin{array}{rcl}
\mathit{MethodType} & ::= & (\tau_i)_1^n \rightarrow \tau_r \\
\mathit{ResultType} & ::= & \mathit{Type} \mid \mathit{VoidType} \\
\mathit{VoidType} & ::= & \texttt{void}
\end{array}$$

The subtype relation $\tau \preceq \tau'$ is defined following the "can be assigned to" criteria as indicated by the desciption of `checkcast` and `instanceof` in [5]. The following functions will be useful:

$$\begin{array}{rcl}
\mathit{superClass}^*(\perp) & = & \{\,\} \\
\mathit{superClass}^*(cl) & = & \{cl.\mathit{superClass}\} \cup \mathit{superClass}^*(cl.\mathit{superClass})
\end{array} \tag{1}$$

$$\begin{array}{rcl}
\mathit{superInterfaces}^*(\mathit{iface}) & = & \bigcup\{\mathit{superInterfaces}^*(\mathit{iface}_i) \mid \mathit{iface}_i \in \mathit{superInterfaces}(\mathit{iface})\} \cup \\
& & \mathit{superInterfaces}(\mathit{iface})
\end{array} \tag{2}$$

$$\begin{array}{rcl}
\mathit{implements}(\perp) & = & \{\,\} \\
\mathit{implements}(cl) & = & \bigcup\{\mathit{superInterfaces}^*(\mathit{iface}_i) \mid \mathit{iface}_i \in \mathit{implementedInterfaces}(cl)\} \cup \\
& & \mathit{implementedInterfaces}(cl) \cup \mathit{implements}(\mathit{superClass}(cl))
\end{array} \tag{3}$$

We write $cl \in \mathit{superClass}^*(cl')$ instead of "cl' is a subclass of cl". Similarly, we write $\mathit{iface} \in \mathit{superInterfaces}^*(\mathit{interface}')$ and $cl \in \mathit{implements}(\mathit{iface})$ if iface' is a subinterface of iface and if cl implements interface iface, respectively.

The subtype relation $\tau \preceq \tau'$ is defined by the rules in Figure 1.

[3] For clarity, we will use this syntactic representation of types instead of the type structures defined in [4]

[4] Empty arrays can be represented with $(\tau_i)_1^n$ and $n = 0$

$$\frac{\tau \in ComponentType}{\tau \preceq \tau} \qquad \frac{cl' \in superClass^*(cl)}{cl \preceq cl'} \qquad \frac{iface \in implements(cl)}{cl \preceq iface}$$

$$\frac{}{iface \preceq \texttt{Object}} \qquad \frac{iface' \in superInterfaces^*(iface)}{iface \preceq iface'}$$

$$\frac{}{(\texttt{array}\ \tau) \preceq \texttt{Object}} \qquad \frac{\tau \preceq \tau'}{(\texttt{array}\ \tau) \preceq (\texttt{array}\ \tau')} \qquad \frac{iface \in implements(\texttt{Array})}{(\texttt{array}\ \tau) \preceq iface}$$

Figure 1: Subtype Relation Rules

3 Program Structures

3.1 Programs and Packages

The information contained in the JCVM CAP files is abstracted by the framework into a *program structure*. A program P contains a set of packages and each package a set of classes. Classes and packages can be extracted from a program using the interface defined in [4]. We define the following domains:

$$P \in Program \quad p \in Package \quad cl \in Class \quad iface \in Interface$$

A package is uniquely identified by an AID (application identifier):

$$aid \in PackageAID$$

Notation: Names are dropped from CAP files. In order to make the presentation more readable, names - concrete syntax - are introduced when necessary, as in the case of types and exception names.

3.2 Classes, Interfaces, methods and fields

The class, interface, method and field structures can be accessed using the interfaces defined in [4].

3.2.1 Method Lookup

All methods sharing the same signature are assigned the same method id. Given a method id and a class cl, the function $methodLookup(id, cl)$ returns the first method structure with the same method id found in the superclass hierarchy of class cl:

$$methodLookup : MethodID \times Class_\perp \to Method_\perp$$

where

$$\begin{array}{lll} methodLookup(id, cl) = m & \Leftrightarrow & m \in cl.methods \ \wedge \ m.id = id \\ methodLookup(id, cl) = methodLookup(id, cl.superClass) & \Leftrightarrow & \forall m \in cl.methods,\ m.id \neq id \\ methodLookup(id, \perp) = \perp & & \end{array}$$

4 Values and Runtime Data Types

JCVML$_e$ supports two kind of values: *primitive values* and *reference values*. Primitive values consist on numeric values (`byte`, `short` and `int`[5]) and return addresses. Reference values consist of references to class instances and arrays (locations, section 5.1) and the special constant `null`. Java Card virtual machine values can be:

$$\begin{array}{rcl} v \in \mathit{Value} & = & \mathit{PrimitiveValue} \cup \mathit{ReferenceValue} \\ \mathit{PrimitiveValue} & = & \mathit{NumericValue} \cup \mathit{ReturnAddressValue} \\ \mathit{NumericValue} & = & \mathit{ByteValue} \cup \mathit{ShortValue} \cup \mathit{IntegerValue} \end{array}$$

where

$$\mathit{ReferenceValue} = \mathit{Location} \cup \{\texttt{null}\}$$

and *ReturnAddressValue* values are runtime representations of *Address* elements. The Java Card virtual machine types associated to the values defined above are:

$$\mathit{JCVMType} ::= \texttt{b} \mid \texttt{s} \mid \texttt{i} \mid \texttt{r} \mid \texttt{ra}$$

for byte, short, integer, reference and return address values respectively. Note that we are re-using the *OperandType* set: $\mathit{JCVMType} = \mathit{OperandType} \cup \{\texttt{ra}\}$.

Storage is managed by the JCVM in terms of an abstract storage unit called *word*. A word is large enough to hold a value of type `b`, `s`, `r` and `ra`. Two words are large enough to hold a value of type `i`.

We define a series of runtime type checks in order to model the static semantics of the virtual machine (verifier). **Runtime type checks** are performed by the *checkType* function. In addition, the function *nbWords* returns the number of words of a value or a sequence of values and the function *getJCVMtype* gets a Java type and returns the respective JCVM type:

$$\begin{array}{rcl} \mathit{checkType} & : & \mathit{Value} \times \mathit{JCVMType} \rightarrow \mathit{boolean} \\ \mathit{nbWords} & : & \mathit{Value}^* \rightarrow \mathit{boolean} \\ \mathit{getJCVMtype} & : & \mathit{Type} \rightarrow \mathit{JCVMType} \end{array}$$

We use the notation v^t to get the runtime value of type t that represents program constant c.

4.1 Numeric Values

The Java types `byte` and `short` are supported by the JCVM. These types correspond to the `b` and `s` JCVM types. Some JCVM implementations may also support the `int` type. The numeric values supported by the JCVM consist of:

- Type `b` (`byte`), represented as 8-bit signed two's complement integers.
- Type `s` (`short`), represented as 16-bit signed two's complement integers.
- Type `i` (`int`), represented as 32-bit signed two's complement integers.

Values of the Java type `bool` are implemented as short values, where 1 represnts `true` and 0 `false`.

Operations between numeric values are performed using the functions *applyUnary* and *applyBinary*:

$$\begin{array}{rcl} \mathit{applyUnary} & : & \mathit{UnaryNumericOperator} \times \mathit{Value} \times \mathit{JCVMType} \times \mathit{JCVMType} \\ \mathit{applyBinary} & : & \mathit{BinaryNumericOperator} \times \mathit{Value} \times \mathit{Value} \times \mathit{JCVMType} \end{array}$$

The *applyUnary* function requires as arguments the operand and result types.The result type is needed by the type conversion operator `to`. Binary operations do not perform type conversions and, therefore, operand and result types are the same.

[5]Not all JCVM implementations support integers.

As mentioned in section 5.3.1, `bool` and `byte` values have to be sign-extended before they are pushed to the operand stack. Similarly, `short` values placed on the stack have to be truncated before they are stored in to a field or array element of type `bool` or `byte`. The *toShort* and *fromShort* functions are defined such that:

$$\begin{array}{lcl} \mathit{toShort}(v) & = & \mathit{applyUnary}(\mathtt{to}, v, t, \mathtt{s}) \ \wedge \ \mathit{checkType}(v, t) \\ \mathit{fromShort}(v, t) & = & \mathit{applyUnary}(\mathtt{to}, v, \mathtt{s}, t) \end{array}$$

5 Run-Time Structures

5.1 Objects and the Heap

Objects are stored in the heap. The heap is defined as a mapping from locations to objects, which can be class instances or arrays.

$$\begin{array}{lcl} H \in \mathit{Heap} & = & \mathit{Location} \ \rightarrow \ \mathit{Object} \\ \mathit{Object} & = & \mathit{ClassInst} \ \cup \ \mathit{ArrayObject} \end{array}$$

where a location is a value of type reference: $\mathit{loc} \in \mathit{Location} \ \Rightarrow \ \mathit{checkType}(\mathit{loc}, \mathtt{r})$.

All objects carry ownership information. This information is kept by a special structure:

$$\mathit{Owner} = (\mathit{context} : \mathit{Context}) \ \times \ (\mathit{aid} : \mathit{AID})$$

where contexts are defined by:

$$\mathit{ctxt} \in \mathit{Context} = \mathit{PackageAID} \ \cup \ \{\mathtt{JCRE}\}$$

A *ClassInst* is defined by:

$$\begin{array}{lcl} o \in \mathit{ClassInst} & = & (\mathit{class} : \mathit{ClassType}) \\ & \times & (\mathit{owner} : \mathit{Owner}) \ \times \ (\mathit{entryPoint} : \mathit{EntryPoint}) \\ & \times & (\mathit{fieldValue} : \mathit{FieldID} \ \rightarrow \ \mathit{Value}) \\ & \times & (\mathit{isInit} : \mathit{boolean}) \end{array}$$

The ***class*** element stores the object's class. ***fieldValue*** is a map from field ids to runtime values and the ***isInit*** flag is used to know if the object has been initialised or not. ***owner*** contains information about the owning context and applet, while ***entryPoint*** indicates if the object is a permanent or temporary `JCRE` entry point[6]:

$$\mathit{EntryPoint} = \{\mathtt{perm}, \mathtt{temp}, \mathtt{no}\}$$

The function *def* maps types to their default initial values. Numeric types map to 0 and reference types to `null`.

$$\mathit{def}(\tau) = \begin{cases} 0 & \text{if } \tau \in \mathit{NumericType} \\ \mathtt{null} & \text{if } \tau \in \mathit{ReferenceType} \end{cases}$$

[6]We assume that being a permanent or temporary entry point is mutually exclusive

New objects are created and allocated by the function *newObject*:

$$\begin{array}{l}
newObject : Owner \times ClassType \times EntryPoint \times Heap \ \rightarrow \ Heap \times Location \\
newObject(own, \tau, ep, H) = < H', loc > \text{ where} \\
\quad \forall loc' \in dom(H).\ H(loc') = H'(loc') \ \wedge \\
\quad H'(loc) = o \ \wedge \ o \in ClassInst \ \wedge \ o.class = \tau \\
\quad o.\text{owner} = own \ \wedge \ o.entryPoint = ep \\
\quad dom(o.fields) = \{f.id \mid f \in instanceFields(\tau.class)\} \ \wedge \ o.isInit = \texttt{false} \\
\quad \forall f \in instanceFields(\tau.class), \\
\qquad o.fieldValue(f.id) = \begin{cases} def(f.type) & \text{if } f.initValue = \perp \\ loc' & \text{if } f.initValue \neq \perp \ \wedge \ f.type = (\texttt{array } \tau') \\ & \text{where} H'(loc') = newDefaultArray(own, \tau', f.initValue) \\ f.initValue & \text{otherwise} \end{cases}
\end{array}$$

where *instanceFields* is the set of non-static fields of a class, including the fields defined in its superclasses. *instanceFields* is defined inductively:

$$\begin{array}{l}
instanceFields(\perp) = \{\ \} \\
instanceFields(cl) = \{f \mid f \in cl.fields \ \wedge \ \neg f.isStatic\} \cup instanceFields(cl.superClass)
\end{array}$$

Arrays can be designated as transient objects. There are two types of transient objects, namely, `CLEAR_ON_RESET` and `CLEAR_ON_DESELECT`. We define:

$$Transient = \{\texttt{CLEAR\_ON\_RESET}, \texttt{CLEAR\_ON\_DESELECT}, \texttt{NOT\_TRANSIENT}\}$$

An *ArrayObject* is defined by:

$$\begin{array}{rcl}
a \in ArrayObject = & & (elementType : Type) \\
& \times & (length : integer) \\
& \times & (owner : Owner) \ \times \ (global : boolean) \\
& \times & (transient : Transient) \\
& \times & (values : integer \ \rightarrow \ Value)
\end{array}$$

New arrays are created and allocated with the *newArray* function:

$$\begin{array}{l}
newArray : Owner \times Type \times integer \times boolean \times Transient \times Heap \ \rightarrow \ Heap \times Location \\
newArray(own, \tau, n, g, t, H) = < H', loc > \text{ where} \\
\quad \forall loc' \in dom(H).\ H(loc') = H'(loc') \ \wedge \\
\quad H'(loc) = a \ \wedge \ a \in ArrayObject \ \wedge \ a.elementType = \tau \\
\quad a.\text{owner} = own \ \wedge \ a.global = g \ \wedge \ a.transient = t \\
\quad dom(a.\text{values}) = \{0, \ldots, n-1\} \\
\quad a.length = n \ \wedge \ \forall i.\ 0 \leq i < n, a.\text{values}(i) = def(\tau)
\end{array}$$

and arrays with a default initial value are created with the *newDefaultArray* function:

$$\begin{array}{l}
newDefaultArray : Owner \times Type \times [integer] \times \ \rightarrow \ ArrayObject \\
newArray(own, \tau, default) = a \text{ where} \\
\quad a.elementType = \tau \ \wedge \ a.owner = own \ \wedge \ a.global = \texttt{false} \ \wedge \ a.transient = \texttt{NOT_TRANSIENT} \\
\quad n = default.length \ \wedge \ dom(a.\text{values}) = \{0, \ldots, n-1\} \\
\quad a.length = n \ \wedge \ \forall i.\ 0 \leq i < n, a.\text{values}(i) = default[i]
\end{array}$$

$$\frac{\mathit{checkType}(v, \mathtt{b})}{H \vdash v \ : \ \mathtt{byte}} \quad \frac{\mathit{checkType}(v, \mathtt{s})}{H \vdash v \ : \ \mathtt{short}} \quad \frac{\mathit{checkType}(v, \mathtt{i})}{H \vdash v \ : \ \mathtt{int}} \quad \frac{\mathit{checkType}(v, \mathtt{ra})}{H \vdash v \ : \ \mathtt{ret}}$$

$$\frac{\begin{array}{l}\mathit{checkType}(loc, \mathtt{r}) \\ H(loc) = a \in \mathit{ArrayObject} \\ \tau = a.\mathit{elementType}\end{array}}{H \vdash loc \ : \ (\mathtt{array}\ \tau)} \quad \frac{\begin{array}{l}\mathit{checkType}(loc, \mathtt{r}) \\ H(loc) = o \in \mathit{ClassInst} \\ \tau.\mathit{class} = o.\mathit{class}\end{array}}{H \vdash loc \ : \ \tau}$$

Figure 2: Runtime Typing Rules

5.2 Runtime Typing

The judgement $H \vdash v \ : \ \tau$ assigns the Java type τ to the stack value v. Primitive types are assigned by just matching the JCVM runtime types to the respective Java type (Section 2.2). Reference types are assigned by inspecting the class instance or array and determining their class or element type.

Definition 1 (Runtime Type Relation) *The runtime type relation $H \vdash v \ : \ \tau$ is defined by the rules given in Figure 2.*

The judgement $H \vdash v \ :\preceq \ \tau$ is used to indicate that runtime value v can by assigned to a Java type τ. In the general case, $H \vdash v \ :\preceq \ \tau$ is true if the runtime type of v is a subtype of τ. In the case that v is a short runtime value the relation is more flexible and allows τ to be any one-word numeric type or boolean. This is in response to the fact that one-word numeric values are sign-extended to values of type short whenever they are pushed on to the stack. The special null reference can be assigned to any reference type.

Definition 2 *We define $H \vdash v \ :\preceq \ \tau$ (A value v can be assigned to a type τ) in the following way:*

$$H \vdash v \ :\preceq \ \tau \quad \Leftrightarrow \quad \begin{cases} \tau \in \mathit{ReferenceType} & \text{if } v = \mathtt{null} \\ \tau \in \{\mathtt{bool}, \mathtt{byte}, \mathtt{short}\} & \text{if } \mathit{checkType}(v, \mathtt{s}) \\ H \vdash v \ : \ \tau' \ \wedge \ \tau' \preceq \tau & \text{otherwise} \end{cases}$$

5.3 Frames

A frame contains information about the state of execution of a method. It is defined as follows:

$$\begin{array}{l} F \in \mathit{Frame} = \mathit{Owner} \times \mathit{Method} \times \mathit{Address} \times \mathit{LocalVar} \times \mathit{OperandStack} \\ F ::= <\mathit{own}, m, pc, V, S> \end{array}$$

A frame contains the following components:

- The owning context and applet, $\mathit{own.context}$ and $\mathit{own.aid}$, respectively.
- The current method structure $m \in \mathit{Method}$.
- The program counter register $pc \in \mathit{Address}$, which contains the address of the instruction currently being executed in method m.
- The array of local variables V(or a map $\mathbb{N} \rightarrow \mathit{Word}$).
- The operand stack S.

5.3.1 Operand Stack

JCVML$_e$ instructions take their dynamic operands from a special structure, the operand stack[7]. The operand stack is defined as a sequence of values:

$$S = v_1 :: \ldots :: v_n \in \mathit{OperandStack} = \mathit{StackValue}^*$$

where values on top of the stack appear on the right-hand side of the sequence i.e. v_n is on top of the stack above. The empty stack is represented with the symbol ϵ. An element can be added to a stack with the **::** operator (S::v) and two stack can be concatenated with the **:** operator (S:S'). For clarity we will write v_1::v_2 instead of ϵ::v_1::v_2.

The operand stack stores values of all types except those of type `byte` or `boolean`. Instead, the Java Card virtual machine converts - sign-extends - `byte` and `boolean` values to `short` values when they are pushed to the operand stack. The domain of stack values is defined as:

$$\mathit{StackValue} = \mathit{Value} - \mathit{ByteValue} \qquad \mathit{StackType} = \mathit{JCVMType} - \{\mathtt{b}\}$$

The stack can also be viewed as a sequence of words:

$$S = w_1 :: \ldots :: w_n \in \mathit{OperandStack} = \mathit{Word}^*$$

We currently use the first representation of stacks - though both can be used - and determine the number of words in the a stack S with the function $\mathit{nbWords}(S)$.

Stack operations do not carry type information, they only require that the integrity of the values is mantained i.e. multiword values must not be split apart.

5.3.2 Local Variables

A local variable is not referenced by name but by the position it occupies in the array of local variables, defined by:

$$V \in \mathit{LocalVar} = [\mathit{StackValue}_\perp]$$

The array of local variables can store values of any of the stack types. Local variable slots are not statically typed so values of different types can be stored in the same slot at different times during the execution of a method. Special care must be taken when using multiword values. An integer value stored in $V[i]$ takes two slots and access to $V[i+1]$ is prohibited. We can model this by making $V[i+1] = \perp$ while position i holds the integer value.

Notation: We write:

$$V = v_0 :: \ldots :: v_n$$

when local variable array V is initialized with the sequence of values (usually from the operand stack) $v_0 :: \ldots :: v_n$. Note that $V[0] = v_0$ but v_i does not necessarily correspond to $V[i]$. Instead we have:

$$V[0] = v_0 \quad \wedge \quad V[\mathit{nbWords}(v_0 :: \ldots :: v_{i-1})] = v_i \quad i \in \{1, \ldots, n\}$$

5.4 Configurations

A runtime configuration describes the state of execution of the JCVM. We define three kinds of configurations:

$$C \in \mathit{Config} \;=\; \mathit{RConfig} \,\cup\, \mathit{EConfig} \,\cup\, \mathit{HConfig}$$

[7] Some instructions also take dynamic operands from the local variables.

A running configuration - *RConfig* - keeps track of the chain of invoked methods by using a stack of call frames. An exception configuration - *EConfig*- represents the state of uncaught exceptions or machine errors. The halt configuration is the result of returning from the method that started the execution of the current applet[8].

A **running configuration** $C = <K,H,SF>$ contains the following components:

- The heap H, a mapping from locations to runtime objects (class and array instances).
- Static fields memory K, a mapping $\mathit{FieldID} \to \mathit{Value}$.
- The call stack SF, where $SF \in \mathit{Frame}^*$.

The frame of the method currently being executed - the current method - sits on top of the call stack. Given $SF = SF' :: <own, m, pc, V, S>$, the current instruction I can be extracted in the following way:

$$I = m.instructionAt(pc)$$

The next instruction can be found at address $m.nextAddress(pc)$.

Exception and halt configurations represent terminal configurations. They are defined by:

$$\begin{array}{rcl} <H,K,loc> \in \mathit{EConfig} & = & \mathit{Heap} \times \mathit{StaticMem} \times \mathit{Location} \\ <H,K,\mathtt{halt}> \in \mathit{HConfig} & = & \mathit{Heap} \times \mathit{StaticMem} \times \{\mathtt{halt}\} \end{array}$$

where the location *loc* in the exception configuration is a reference to the uncaught exception object.

6 Exceptions

An **exception** is thrown whenever a program violates the semantic constraints of JCVMLe or any other constraint specified by the programmer. Exceptions can be thrown explicitly using the `throw` instruction.

An exception is said to be caught if an appropiate **exception handler** is found in the current stack of call frames. If an exception is caught, control is transfered to the start address indicated by the exception handler and execution is resumed at that point. An uncaught exception will cause the virtual machine to halt.

The process of throwing and catching an exception is defined by the *catchException* function:

$$\mathit{catchException} : \mathit{Frame}^* \times \mathit{Heap} \times \mathit{Location} \to \mathit{Frame}^* \cup \mathit{Location}$$

Let $H \vdash loc \ : \ \tau$

$$\begin{array}{rcl} \mathit{catchException}(SF{::}<own, m, pc, V, S>,H,loc) & = & \mathit{catchException}(SF, H, loc) \\ & & \quad \text{if } \bot = \mathit{findHandler}(m, pc, \tau) \\ \mathit{catchException}(SF{::}<own, m, pc, V, S>,H,loc) & = & SF{::}<own, m, pc', V, loc> \\ & & \quad \text{if } pc' = \mathit{findHandler}(m, pc, H, loc) \ \wedge \ pc' \neq \bot \\ \mathit{catchException}(\epsilon, H, loc) & = & loc \end{array}$$

The *catchException* function searches the stack of call frames according to the following criteria:

- It first inspects the top call frame to see if the exception of class τ and referenced by *loc* can be handled. If no appropiate exception handler is found in the top frame, the frame is popped and the search continues from the invoker's frame ("...frame is popped, the frame of its invoker is reinstated, and *loc* is rethrown"[6, Chapter 7, page 133]).

[8] We can also use the halt configuration to model uncaught exceptions

- If the top frame contains an appropiate handler, the *pc* is reset to the value indicated by the handler, the operand stack of the frame is cleared and *loc* is pushed back onto the operand stack and the new stack of frames is returned.
- If no frame is found that can handle the exception - if the stack of frames is the empty stack - the function returns the reference to the uncaught exception.

The $\mathit{findHandler}(m, pc, \tau)$ function searches for the **first** exception handler in method m that can handle an exception of class τ at address pc. It returns the address of the code intended to handle the exception.The semantics of *findHandler* is described below.

Let $E = m.\mathit{exceptionHandlers}$:

$$\mathit{findHandler}(m, pc, \tau) = \begin{cases} E[i].\mathit{handlerAddress} & \exists i \in \{0, \ldots, E.\mathit{length} - 1\}. \\ & \quad \mathit{isCorrectHandler}(E, i, pc, \tau) \wedge \\ & \quad \forall j, 0 \leq j < i : \neg \mathit{isCorrectHandler}(E, j, pc, \tau) \\ \perp & \text{otherwise} \end{cases}$$

The predicate $\mathit{isCorrectHandler}(E, i, pc, \tau)$ determines if exception handler $E[i]$ can handle an exception τ at address pc:

$$\mathit{isCorrectHandler}(E, i, pc, \tau) = \begin{cases} \texttt{true} & (E[i].\mathit{startAddress} \leq pc \leq E[i].\mathit{endAddress}) \wedge \\ & (\tau \preceq E[i].\mathit{catchType} \vee E[i].\mathit{catchType} = \perp) \\ \texttt{false} & \text{otherwise} \end{cases}$$

It is the job of the compiler to order the table in such a way that, given a set of nested exceptions, the innermost exceptions appear first.

6.1 Runtime Exceptions

Runtime exceptions maybe raised by the JCVM during the execution of a program. These are `JCRE` owned, pre-allocated exceptions designated as temporary `JCRE` Entry Point Objects that can be requested with the *getExceptionLocation* function:

$$\mathit{getExceptionLocation} : \mathit{ClassType} \rightarrow \mathit{Location}$$

We define a set of predicates that determine the conditions for throwing an exception.

$\mathit{checkNull}(loc)$	$\equiv$	$(loc \neq \texttt{null})$	`NullPointerException`
$\mathit{checkArithmetic}(op, v_1, v_2)$	$\equiv$	$op \notin \{\texttt{div}, \texttt{rem}\} \vee v_2 \neq 0$	`ArithmeticException`
$\mathit{checkNegative}(n)$	$\equiv$	$(n \geq 0)$	`NegativeArraySizeException`
$\mathit{checkCanCast}(H, loc, \tau)$	$\equiv$	$H \vdash loc :\preceq \tau$	`ClassCastException`
$\mathit{checkArrayBounds}(a, i)$	$\equiv$	$(0 \leq i < a.\mathit{length})$	`ArrayIndexOutOfBoundsException`

7 Static Semantics and Verification

Each operational semantics rule includes a boxed area labelled as "Verification" containing a set of conditions that must be certified by the verifier[9]. These set of conditions - which must be true since we assume that the code has been verified - will be our starting point to define the static semantics of the machine: A verifier must make sure that these conditions hold for every state of execution. Right now, we have only included (most of) the conditions regarded as "must" conditions by the specification, which are type related or deal with stack sizes (number of words).

[9]The verifier implements the static semantics of the machine.

8 The JCVM Firewall

Arrays and class instances are owned by the applet that created them. The context (or package) where the applet is defined determines the owning context. The JCVM firewall provides a security mechanism that prevents an object from being accessed by code running in a different context (different from the owning context).

Prior to performing an operation on an object, the JCVM performs an access check. The access checks depend on the type and owner of the referenced object, the instruction, and the currently active context. A good description of the access checks can be found in [1]. We define the following access check predicates:

$$\begin{array}{rcl}
\mathit{checkPutStatic} & : & \mathit{Context} \times \mathit{Heap} \times \mathit{Value} \rightarrow \mathit{boolean} \\
\mathit{checkGetField} & : & \mathit{Context} \times \mathit{Object} \rightarrow \mathit{boolean} \\
\mathit{checkPutField} & : & \mathit{Context} \times \mathit{Object} \times \mathit{Heap} \times \mathit{Value} \rightarrow \mathit{boolean} \\
\mathit{checkThrow} & : & \mathit{Context} \times \mathit{Object} \rightarrow \mathit{boolean} \\
\mathit{checkInvokeDefinite} & : & \mathit{Context} \times \mathit{Object} \rightarrow \mathit{boolean} \\
\mathit{checkInvokeVirtual} & : & \mathit{Context} \times \mathit{Context} \times \mathit{Object} \rightarrow \mathit{boolean} \\
\mathit{checkInvokeIface} & : & \mathit{Context} \times \mathit{Context} \times \mathit{Interface} \times \mathit{Object} \rightarrow \mathit{boolean} \\
\mathit{checkCast} & : & \mathit{Context} \times \mathit{Context} \times \mathit{ReferenceType} \times \mathit{Object} \rightarrow \mathit{boolean} \\
\mathit{checkArrayLoad} & : & \mathit{Context} \times \mathit{Context} \times \mathit{ArrayObject} \rightarrow \mathit{boolean} \\
\mathit{checkArrayStore} & : & \mathit{Context} \times \mathit{Context} \times \mathit{ArrayObject} \times \mathit{Heap} \times \mathit{Value} \rightarrow \mathit{boolean}
\end{array}$$

Access to objects is restricted by the JCVM firewall. In general, a security exception is thrown if the current context - the context of the current frame - is different from the owning context of the object that is being referenced, unless it is the `JCRE` context, which has special privileges. Given current context $ctxt$ and object o, access is allowed if:

$$(\mathit{ctxt} = \mathtt{JCRE}) \ \vee \ (\mathit{ctxt} = o.\mathit{ownercontext})$$

The `JCRE` provides several mechanisms to allow object access across contexts. `JCRE` entry point objects - temporary and permanenet - and global arrays are objects owned by the `JCRE` context that can be accessed from any context. A context switch occurs when a method of an object owned by a different context is invoked i.e. two consecutive frames have different contexts. The *anyContext* predicate checks if an object is a global array of if it has been designated as a `JCRE` entry point:

$$\mathit{anyContext}(o) = (o \in \mathit{ClassInst} \ \wedge \ o.\mathit{entryPoint} \neq \mathtt{no}) \ \vee \ (o \in \mathit{ArrayObject} \ \wedge \ o.\mathit{global})$$

Objects designated as temporary `JCRE` entry points and global arrays cannot be stored in class variables, instance variables or array components. Let $\tau_r \in \mathit{ReferenceType}$. The *isStorable* predicate is defined as:

$$\mathit{isStorable}(H, v) = [H \vdash v \ : \ \tau_r \ \wedge \ o = H(v)] \ \Rightarrow \ \left\{ \begin{array}{l} (o \in \mathit{ClassInst} \ \wedge \ o.\mathit{entryPoint} \neq \mathtt{temp}) \ \vee \\ (o \in \mathit{ArrayObject} \ \wedge \ \neg o.\mathit{global}) \end{array} \right.$$

Access to static and instance fields is checked by the following predicates:

$$\begin{array}{rcl}
\mathit{checkPutStatic}(\mathit{ctxt}, H, v) & \equiv & \mathit{ctxt} = \mathtt{JCRE} \ \vee \ \mathit{isStorable}(H, v) \\
\mathit{checkGetField}(\mathit{ctxt}, o) & \equiv & \mathit{ctxt} = \mathtt{JCRE} \ \vee \ \mathit{ctxt} = o.\mathit{owner.context} \\
\mathit{checkThrow}(\mathit{ctxt}, o) & \equiv & \mathit{checkGetField}(\mathit{ctxt}, o) \\
\mathit{checkPutField}(\mathit{ctxt}, o, H, v) & \equiv & \mathit{ctxt} = \mathtt{JCRE} \ \vee \ (\mathit{ctxt} = o.\mathit{owner.context} \ \wedge \ \mathit{isStorable}(H, v))
\end{array}$$

Arrays with primitive type components or references to `Object` can be transient objects. Transient objects of `CLEAR_ON_DESELECT` type can only be accessed when the currently active context is the context of the currently selected apple. The *checkTransient* predicate checks for `CLEAR_ON_DESELECT` transient objects:

$$\mathit{checkTransient}(\mathit{ctxt}, \mathit{appCtxt}, o) \equiv \left\{ \begin{array}{ll} (o.\mathit{transient} = \mathtt{CLEAR\_ON\_DESELECT}) \ \Rightarrow \ (\mathit{ctxt} = \mathit{appCtxt}) & o \in \mathit{ArrayObject} \\ \mathtt{true} & o \in \mathit{ClassInst} \end{array} \right.$$

Access to array elements is checked by the following predicates:

$$\mathit{checkArrayLoad}(\mathit{ctxt}, \mathit{appCtxt}, a) \quad \equiv \quad \mathit{ctxt} = \texttt{JCRE} \vee \left(\begin{array}{c} (\mathit{ctxt} = a.\mathit{owner}.\mathit{context} \ \vee \ a.\mathit{global}) \wedge \\ \mathit{checkTransient}(\mathit{ctxt}, \mathit{appCtxt}, a) \end{array} \right)$$

$$\mathit{checkArrayStore}(\mathit{ctxt}, \mathit{appCtxt}, a, H, v) \quad \equiv \quad \mathit{ctxt} = \texttt{JCRE} \vee \left(\begin{array}{c} (\mathit{ctxt} = a.\mathit{owner}.\mathit{context} \ \vee \ a.\mathit{global}) \wedge \\ \mathit{checkTransient}(\mathit{ctxt}, \mathit{appCtxt}, a) \wedge \\ \mathit{isStorable}(H, v) \end{array} \right)$$

checkInvokeDefinite follows the same restrictions for contexts but takes into consideration the invocation of methods on objects designated as entry points - *anyContext* is used. Putting everything together we have:

$$\mathit{checkInvokeDefinite}(\mathit{ctxt}, o) \quad \equiv \quad (\mathit{ctxt} = \texttt{JCRE}) \ \vee \ (\mathit{ctxt} = o.\mathit{owner}.\mathit{context}) \ \vee \ \mathit{anyContext}(o)$$

It is possible to invoke methods of the `Object` class on an array. In particular, methods can be invoked on a transient array. The *checkTransient* check must be added to the security checks for `invokevirtual`:

$$\mathit{checkInvokeVirtual}(\mathit{ctxt}, \mathit{appCtxt}, o) \quad \equiv \quad \mathit{ctxt} = \texttt{JCRE} \vee \left(\begin{array}{c} [(\mathit{ctxt} = o.\mathit{owner}.\mathit{context}) \ \vee \ \mathit{anyContext}(o)] \\ \wedge \\ \mathit{checkTransient}(\mathit{ctxt}, \mathit{appCtxt}, o)] \end{array} \right)$$

Another mechanism that allows object access among contexts is the use of shareable interfaces. Methods of shareable interfaces can be invoked from one context even if the object implementing them is owned by an applet in another context. The following must be true:

$$\mathit{isShareable}(o, \tau) \equiv \texttt{Shareable} \in (\mathit{implements}(o.\mathit{class}) \cap \mathit{superInterfaces}^*(\mathit{iface}))$$

The security check for the invoke interface instruction is defined by:

$$\begin{array}{l} \mathit{checkInvokeIface}(\mathit{ctxt}, \mathit{appCtxt}, \mathit{iface}, o) \equiv \\ \qquad (\mathit{ctxt} = \texttt{JCRE}) \ \vee \ \left(\begin{array}{c} [(\mathit{ctxt} = o.\mathit{owner}.\mathit{context}) \ \vee \ \mathit{anyContext}(o) \ \vee \ \mathit{isShareable}(o, \tau)] \\ \wedge \\ \mathit{checkTransient}(\mathit{ctxt}, \mathit{appCtxt}, o) \end{array} \right) \end{array}$$

Security checks for `checkcast` and `instanceof` are performed by:

$$\begin{array}{l} \mathit{checkCast}(\mathit{ctxt}, \mathit{appCtxt}, \tau, o) \equiv \\ \left\{ \begin{array}{l} (\mathit{ctxt} = \texttt{JCRE}) \vee \\ \left(\left\{ \begin{array}{l} (\mathit{ctxt} = o.\mathit{owner}.\mathit{context}) \ \vee \ \mathit{anyContext}(o) \vee \\ (\tau \in \mathit{InterfaceType} \ \wedge \ \mathit{isShareable}(o, \tau.\mathit{interface})) \end{array} \right\} \wedge \ \mathit{checkTransient}(\mathit{ctxt}, \mathit{appCtxt}, o) \right) \end{array} \right. \end{array}$$

9 The JCVML$_e$ Operational Semantics

The small step semantics is the smallest relation *Config* $\Rightarrow$ *Config* defined by the rules below.

9.1 The Core Language [C1]

NoOperationInstruction

Instruction	`nop`
Short Description	No operation
JCVM bytecodes	*nop*

Description: Do Nothing.

Rule:

$$\frac{I = m.instructionAt(pc) \ \wedge \ I = \texttt{nop}}{<K,H,SF::<own,\, m,\, pc,\, V,\, S>> \ \Rightarrow \ <K,H,SF::<own,\, m,\, m.nextAddress(pc),\, V,\, S>>}$$

Notation: For convenience, from now on we will consider $I = m.instructionAt(pc)$ as the default and write 0 and $pc + 1$ instead of $firstAddress(m)$ and $m.nextAddress(pc)$, respectively, whenever m and pc are clear from the context. Then we have:

$$\frac{I = \texttt{nop}}{<K,H,SF::<own,\, m,\, pc,\, V,\, S>> \ \Rightarrow \ <K,H,SF::<own,\, m,\, pc+1,\, V,\, S>>} \quad \text{(R 1.1)}$$

PushConstantInstruction

Instruction	`push` t c
Short Description	Push constant c of type t
JCVM bytecodes	*aconst_null, sconst_<s>, iconst_<s>, {b,s}spush, {b,s,i}ipush*

$$\frac{I = \texttt{push} \ t \ c}{<K,H,SF::<own,\, m,\, pc,\, V,\, S>> \ \Rightarrow \ <K,H,SF::<own,\, m,\, pc+1,\, V,\, S::c^t>>} \quad \text{(R 2.1)}$$

Static Constraints: $c \ : \ \tau$

PopWordsInstruction

Instruction	`pop` n
Short Description	Pop n top operand stack words
JCVM bytecodes	*pop, pop2*

Description: The top words are popped from the operand stack.

$$\frac{I = \texttt{pop} \ n \ \wedge \ n = nbWords(S')}{<K,H,SF::<own,\, m,\, pc,\, V,\, S{:}S'>> \ \Rightarrow \ <K,H,SF::<own,\, m,\, pc+1,\, V,\, S>>} \quad \text{(R 3.1)}$$

Notes: The pop instruction operates on untyped words, ignoring the type of data they represent.

DuplicateWordsInstruction

Instruction	`dup` n d
Short Description	Duplicate words $1 \ldots n$ at depth d
JCVM bytecodes	*dup,dup_x, dup2*

$$\frac{\begin{array}{l} I = \texttt{dup} \ n \ d \\ S = S_3 : S_2 : S_1 \\ n = nbWords(S_1) \ \wedge \ d = nbWords(S_2 : S_1) \\ S' = S_3 : S_1 : S_2 : S_1 \end{array}}{<K,H,SF::<own,\, m,\, pc,\, V,\, S>> \ \Rightarrow \ <K,H,SF::<own,\, m,\, pc+1,\, V,\, S'>>} \quad \text{(R 4.1)}$$

Description[6]: The top n words on the operand stack are duplicated and the copied words are inserted d words down in the operand stack. When d equals 0, the top n words are copied and placed on top of the stack.

SwapWordsInstruction

Instruction	`swap` n_1 n_2
Short Description	Swap words $1 \ldots n_1$ with words $(n_1+1)\ldots(n_1+n_2)$
JCVM bytecodes	*swap_x*

$$\frac{\begin{array}{l} I = \texttt{swap}\ \ n_1\ \ n_2 \\ S = S_3 : S_2 : S_1 \\ n_1 = nbWords(S_1)\ \wedge\ n_2 = nbWords(S_2) \\ S' = S_3 : S_1 : S_2 \end{array}}{<K,H,SF::<own, m, pc,\ V,\ S>>\ \Rightarrow\ <K,H,SF::<own, m, pc+1,\ V,\ S'>>} \quad \text{(R 5.1)}$$

Description: The top n_1 words on the operand stack are swapped with the n_2 words immediately below.

NumericOperationInstruction

Instruction	`numop` t op t'
Short Description	Numeric Operation op, operands typed t, result t'
JCVM bytecodes	...,*sshl, sshr, sushr, sand, sor, sxor*

- Binary Operations:

$$\frac{\begin{array}{l} I = \texttt{numop}\ \ t\ \ op\ \ t' \\ op \in BinaryNumericOperator \\ checkArithmetic(op, v_1, v_2) \\ r = applyBinary(op, v_1, v_2) \end{array}}{<K,H,SF::<own, m, pc,\ V,\ S::v_1::v_2>>\ \Rightarrow\ <K,H,SF::<own, m, pc+1,\ V,\ S::r>>} \quad \text{(R 6.1)}$$

Verification: $\boxed{checkType(v_1, t)\ \wedge\ checkType(v_2, t)}$

Runtime exception:

$$\frac{\begin{array}{l} I = \texttt{numop}\ \ t\ \ op\ \ t' \\ \neg checkArithmetic(op, v_1, v_2) \\ exception = \texttt{ArithmeticException} \\ loc = getExceptionLocation(exception) \\ SF' = catchException(SF :: <own, m, pc,\ V,\ S::v_1::v_2>, H, loc) \end{array}}{<K,H,SF::<own, m, pc,\ V,\ S::v_1::v_2>>\ \Rightarrow\ <K,H,SF'>} \quad \text{(R 6.2)}$$

- Unary Operations:

$$\frac{\begin{array}{l} I = \texttt{numop}\ \ t\ \ op\ \ t' \\ op \in UnaryNumericOperator \\ r = applyUnary(op, v, t') \end{array}}{<K,H,SF::<own, m, pc,\ V,\ S::v>>\ \Rightarrow\ <K,H,SF::<own, m, pc+1,\ V,\ S::r>>} \quad \text{(R 6.3)}$$

Verification: $\boxed{checkType(v,t)}$

Notes: Despite the fact that overflow or underflow may occur, in which case the result may have a different sign from the true mathematical result, execution of the instruction **never throws a runtime exception**, except as noted above.

LoadLocalVariableInstruction

Instruction	`load` t i
Short Description	Load from local variable of type t at index i
JCVM bytecodes	*{a,s,i}load[_<n>]*

$$\frac{I = \texttt{load}\ t\ i}{<K,H,SF::<own,m,pc,V,S>> \ \Rightarrow\ <K,H,SF::<own,m,pc+1,V,S::V[\mathrm{i}]>>} \quad \text{(R 7.1)}$$

Verification: $\boxed{checkType(V[i],t)}$

Description: i must be a valid index into the local variables of the current frame. The value of the local variable at i is pushed on top of the operandstack.

StoreLocalVariableInstruction

Instruction	`store` t i
Short Description	Store into local variable of type t at index i
JCVM bytecodes	*{a,s,i}store[_<n>]*

$$\frac{I = \texttt{store}\ t\ i}{<K,H,SF::<own,m,pc,V,S::v>> \ \Rightarrow\ <K,H,SF::<own,m,pc+1,V[i\ \mapsto\ v],S>>} \quad \text{(R 8.1)}$$

Verification: $\boxed{checkType(v,\tau)}$

Description: The value v is popped from the top of the operand stack, and the value of the local variable at i is set to v.

IncrementLocalVariableInstruction

Instruction	`inc` t i c
Short Description	Increment local variable of type t at index i by constant c
JCVM bytecodes	*sinc[_w]* (*iinc[_w]*)

$$\frac{\begin{array}{c} I = \texttt{inc}\ t\ i\ c \\ r = V[i] + c \end{array}}{<K,H,SF::<own,m,pc,V,S>> \ \Rightarrow\ <K,H,SF::<own,m,pc+1,V[i\ \mapsto\ r],S>>} \quad \text{(R 9.1)}$$

Verification: $\boxed{checkType(V[\mathrm{i}],t)}$

Syntax: $t \in \{\mathtt{s},\mathtt{i}\}$

UnconditionalBranchInstruction

Instruction	`goto` *addr*
Short Description	Branch at *addr*
JCVM bytecodes	*goto[_w]*

$$\frac{I = \mathtt{goto}\ \ addr}{<K,H,SF::<own,m,pc,V,S>> \ \Rightarrow\ <K,H,SF::<own,m,addr,V,S>>} \quad \text{(R 10.1)}$$

Verification: $\boxed{m.instructionAt(addr) \neq \perp}$

Description: Branch always at address *addr*. The target address must be that of an opcode of an instruction within the method that contains the `goto` instruction.

ConditonalJumpInstruction

Instruction	`if` *t* *op* $\mathtt{null}_{opt}$ `goto` *addr*
Short Description	Branch at *addr* if comparison *op* on type *t* succeeds
JCVM bytecodes	*if_{a,s}cmp<cond>[_w], if_<cond>[_w], if[non]null[_w]*

We have the following cases:

- Not null.

$$\frac{\begin{array}{l} I = \mathtt{if}\ \ t\ \ op\ \ \mathtt{goto}\ \ addr \\ pc' = \begin{cases} addr & \text{if } applyBinary(op,v_1,v_2) = \mathtt{true} \\ m.nextAddress(pc) & \text{otherwise} \end{cases} \end{array}}{<K,H,SF::<own,m,pc,V,S::v_1::v_2>> \ \Rightarrow\ <K,H,SF::<own,m,pc',V,S>>} \quad \text{(R 11.1)}$$

  If $t = \mathtt{r}$ then $op \in \{\mathtt{eq},\mathtt{ne}\}$.

  Verification: $\boxed{\begin{array}{l} i = 1,2\ \ checkType(v_i,t) \\ m.instructionAt(addr) \neq \perp \end{array}}$

- Null comparison.

$$\frac{\begin{array}{l} I = \mathtt{if}\ \ t\ \ op\ \ \mathtt{null}\ \ \mathtt{goto}\ \ addr \\ pc' = \begin{cases} addr & \text{if } (t = \mathtt{r}\ \wedge\ applyBinary(op,v,\mathtt{null}) = \mathtt{true}) \vee \\ & \quad (t = \mathtt{s}\ \wedge\ applyBinary(op,v,0) = \mathtt{true}) \\ m.nextAddress(pc) & \text{otherwise} \end{cases} \end{array}}{<K,H,SF::<own,m,pc,V,S::v>> \ \Rightarrow\ <K,H,SF::<own,m,pc',V,S>>} \quad \text{(R 11.2)}$$

  where $t = \{\mathtt{r},\mathtt{s}\}$ and $op \in \{\mathtt{eq},\mathtt{ne}\}$.

  Verification: $\boxed{\begin{array}{l} checkType(v,t) \\ m.instructionAt(addr) \neq \perp \end{array}}$

Description: If the null comparison keyword is not present then both v_1 and v_2 are popped from the operand stack and compared, depending on the value of the comparison operator *op*. If the comparison succeeds, *addr* is used as the new *pc*, otherwise, execution proceeds at the address of the instruction following the current `if` instruction. If the operand type t is a reference then v_1 and v_2 are tested for equality.

If the null comparison keyword is present then only v is popped from the operand stack and compared against `null` (or 0) for equality. If the comparison succeeds, *addr* is used as the new *pc*, otherwise, execution proceeds at the address of the instruction following the current `if` instruction.

Restriction: The target address *pc*' must be that of an opcode of an instruction within the method that contains the current instruction.

KeySwitchInstruction

Instruction	`lookupswitch` t $(k$ `=>` $addr)^*$, `default =>` $addr'$
Short Description	Branch at $addr_i$ if operand $= k_i$, otherwise at *addr*'
JCVM bytecodes	*slookupswitch* (*ilookupswitch*)

$$\frac{\begin{array}{l} I = \texttt{lookupswitch}\ t\ \ (k_i \texttt{ => } addr_i)_1^n\ \texttt{, default => }\ addr' \\ a = \begin{cases} addr_i & \text{if } \exists i \in \{1,\ldots,n\}\ key = k_i \\ addr' & \text{otherwise} \end{cases} \end{array}}{<K,H,SF::<own,m,pc,V,S::key>> \ \Rightarrow\ <K,H,SF::<own,m,a,V,S>>} \quad \text{(R 12.1)}$$

where $t \in \{\mathtt{s}, \mathtt{i}\}$.

Verification:

$$\boxed{\begin{array}{l} m.instructionAt(addr_i) \neq \bot \ \wedge \ m.instructionAt(addr') \neq \bot \\ checkType(key, t) \end{array}}$$

IndexSwitchInstruction

Instruction	`tableswitch` t l `=>` $(addr)^*$, `default =>` $addr'$
Short Description	Branch at $addr_i$ if operand $= l + i$, otherwise at *addr*'
JCVM bytecodes	*stableswitch* (*itableswitch*)

$$\frac{\begin{array}{l} I = \texttt{tableswitch}\ t\ \ l \texttt{ => } (addr_i)_0^n\ \texttt{, default => }\ addr' \\ a = \begin{cases} addr_i & \text{if } l \leq index \leq (l+n)\ \wedge\ j = index - l \\ addr' & \text{otherwise} \end{cases} \end{array}}{<K,H,SF::<own,m,pc,V,S::index>> \ \Rightarrow\ <K,H,SF::<own,m,a,V,S>>} \quad \text{(R 13.1)}$$

where $t \in \{\mathtt{s}, \mathtt{i}\}$

Verification:

$$\boxed{\begin{array}{l} m.instructionAt(addr_i) \neq \bot \ \wedge \ m.instructionAt(addr') \neq \bot \\ checkType(index, t) \end{array}}$$

9.2 The Object Language [O2]

NewInstruction

Instruction	`new` τ
Short Description	Created new object of reference type t
JCVM bytecodes	*new, newarray, anewarray*

- For array objects:

$$\frac{\begin{array}{l} I = \texttt{new}\ (\texttt{array}\ \tau) \\ checkNegative(n) \\ <H', loc> = newArray(own, \tau, n, \texttt{false}, \texttt{NOT\_TRANSIENT}, H) \end{array}}{<K,H,SF::<own, m, pc, V, S::n>> \ \Rightarrow\ <K,H',SF::<own, m, pc+1, V, S::loc>>} \quad \text{(R 14.1)}$$

Verification: $\boxed{checkType(n,\texttt{s})}$

Runtime Exceptions:

$$\frac{\begin{array}{l} I = \texttt{new}\ (\texttt{array}\ \tau)\ \wedge\ \neg checkNegative(n) \\ exception = \texttt{NegativeArraySizeException} \\ loc = getExceptionLocation(exception) \\ SF' = catchException(SF::<own, m, pc, V, S::n>, H, loc) \end{array}}{<K,H,SF::<own, m, pc, V, S::n>> \ \Rightarrow\ <K,H,SF'>} \quad \text{(R 14.2)}$$

- For class instances:

$$\frac{\begin{array}{l} I = \texttt{new}\ cl\ \wedge\ cl \in Class \\ <H', loc> = newObject(own, cl, \texttt{no}, H) \end{array}}{<K,H,SF::<own, m, pc, V, S>> \ \Rightarrow\ <K,H',SF::<own, m, pc+1, V, S::loc>>} \quad \text{(R 14.3)}$$

Verification: $\boxed{\neg cl.isAbstract}$

GetStaticFieldInstruction

Instruction	`getstatic` f
Short Description	Get Static field f from class
JCVM bytecodes	*getstatic_<t>*

$$\frac{\begin{array}{l} I = \texttt{getstatic}\ f \\ v = \begin{cases} toShort(K(f.id)) & \text{if } f.type \in \{\texttt{bool}, \texttt{byte}\} \\ K(f.id) & \text{otherwise} \end{cases} \end{array}}{<K,H,SF::<own, m, pc, V, S>> \ \Rightarrow\ <K,H,SF::<own, m, pc+1, V, S::v>>} \quad \text{(R 15.1)}$$

PutStaticFieldInstruction

Instruction	`putstatic` f
Short Description	Set static field f in class
JCVM bytecodes	*putstatic_<t>*

$$\frac{\begin{array}{l} I = \texttt{putstatic}\ f \\ v' = \begin{cases} fromShort(v, \texttt{b}) & \text{if } f.type \in \{\texttt{bool}, \texttt{byte}\} \\ v & \text{otherwise} \end{cases} \\ checkPutStatic(own.context, H, v) \end{array}}{<K,H,SF::<own, m, pc, V, S::v>> \ \Rightarrow\ <K[f.id \mapsto v'],H,SF::<own, m, pc+1, V, S>>} \quad \text{(R 16.1)}$$

Verification: $\boxed{checkType(v, f.type)}$

Runtime Exceptions:

$$\frac{\begin{array}{l} I = \texttt{putstatic}\ f \\ \neg checkPutStatic(own.context, H, v) \\ exception = \texttt{SecurityException} \\ loc = getExceptionLocation(exception) \\ SF' = catchException(SF::<own,\ m, pc,\ V, S::v>, H, loc) \end{array}}{<K,H,SF::<own,\ m, pc,\ V, S::v>> \ \Rightarrow\ <K,H,SF'>} \quad \text{(R 16.2)}$$

GetInstanceFieldInstruction

Instruction	`getfield` $\texttt{this}_{opt}$ f
Short Description	Fetch field f from [current] object
JCVM bytecodes	*getfield_<t>[_this]*

$$\frac{\begin{array}{l} \begin{cases} loc = V[0] \ \wedge\ S' = S & \text{if } I = \texttt{getfield this}\ f \\ S = S' :: loc & \text{if } I = \texttt{getfield}\ f \end{cases} \\ o = H(loc) \ \wedge\ \tau = f.type \\ v = \begin{cases} toShort(o.fieldValue(f.id)) & \text{if } f.type \in \{\texttt{bool}, \texttt{byte}\} \\ o.fieldValue(f.id) & \text{otherwise} \end{cases} \\ checkNull(loc) \ \wedge\ checkGetField(own.context, o) \end{array}}{<K,H,SF::<own,\ m, pc,\ V, S>> \ \Rightarrow\ <K,H,SF::<own,\ m, pc+1,\ V, S'::v>>} \quad \text{(R 17.1)}$$

Verification:

$$\boxed{\begin{array}{l} checkType(loc, \texttt{r}) \ \wedge\ o \in ClassInst \\ \left(\begin{array}{l} f.isProtected \ \wedge \\ f.classI \in superClass^*(m.classI) \ \wedge \\ f.classI.package \neq m.classI.package \end{array}\right) \Rightarrow o.class \preceq m.classI \end{array}}$$

Runtime Exceptions:

$$\frac{\begin{array}{l} \begin{cases} loc = V[0] \ \wedge\ S' = S & \text{if } I = \texttt{getfield this}\ f \\ S = S' :: loc & \text{if } I = \texttt{getfield}\ f \end{cases} \\ exception = \begin{cases} \texttt{NullPointerException} & \text{if } \neg checkNull(loc) \\ \texttt{SecurityException} & \text{if } \neg checkGetField(own.context, H(loc)) \end{cases} \\ loc' = getExceptionLocation(exception) \\ SF' = catchException(SF::<own,\ m, pc,\ V, S>, H, loc') \end{array}}{<K,H,SF::<own,\ m, pc,\ V, S>> \ \Rightarrow\ <K,H,SF'>} \quad \text{(R 17.2)}$$

PutInstanceFieldInstruction

Instruction	`putfield` $\texttt{this}_{opt}$ f
Short Description	Set field f in [current] object
JCVM bytecodes	*putfield_<t>[_this]*

$$\dfrac{\begin{array}{l}\left\{\begin{array}{ll} loc = V[0] \ \wedge \ S' = S & \text{if } I = \texttt{putfield this } f \\ S = S' :: loc & \text{if } I = \texttt{putfield } f \end{array}\right. \\ o = H(loc) \ \wedge \ \tau = f.type \\ v' = \left\{\begin{array}{ll} fromShort(v, \texttt{b}) & \text{if } f.type \in \{\texttt{bool}, \texttt{byte}\} \\ v & \text{otherwise} \end{array}\right. \ \wedge \ \begin{array}{l} fv = o.fieldValue[f.id \mapsto v'] \\ o' = o[fieldValue \mapsto fv] \end{array} \\ checkNull(loc) \ \wedge \ checkPutField(own.context, o, H, v) \end{array}}{<K,H,SF::<own, m, pc, V, S::v>> \ \Rightarrow \ <K,H[loc \mapsto o'],SF::<own, m, pc+1, V, S'>>} \quad \text{(R 18.1)}$$

Verification:

$$\boxed{\begin{array}{l} checkType(loc, \texttt{r}) \quad \wedge \quad o \in ClassInst \\ checkType(v, getJCVMtype(\tau)) \\ \left(\begin{array}{l} f.isProtected \ \wedge \\ f.classI \in superClass^{*}(m.classI) \ \wedge \\ f.classI.package \neq m.classI.package \end{array}\right) \Rightarrow o.class \preceq m.classI \end{array}}$$

Runtime Exceptions:

$$\dfrac{\begin{array}{l}\left\{\begin{array}{ll} loc = V[0] \ \wedge \ S' = S & \text{if } I = \texttt{putfield this } f \\ S = S' :: loc & \text{if } I = \texttt{putfield } f \end{array}\right. \\ exception = \left\{\begin{array}{ll} \texttt{NullPointerException} & \text{if } \neg checkNull(loc) \\ \texttt{SecurityException} & \text{if } \neg checkPutField(own.context, H(loc), H, v) \end{array}\right. \\ loc' = getExceptionLocation(exception) \\ SF' = catchException(SF::<own, m, pc, V, S::v>, H, loc') \end{array}}{<K,H,SF::<own, m, pc, V, S::v>> \ \Rightarrow \ <K,H,SF'>} \quad \text{(R 18.2)}$$

CheckCastInstruction

Instruction	`checkcast` τ
Short Description	Check whether object is of type τ
JCVM bytecodes	*checkcast*

$$\dfrac{\begin{array}{l} I = \texttt{checkcast } \tau \\ loc \neq \texttt{null} \ \Rightarrow \ checkCast(own.context, \tau, H(loc)) \\ checkCanCast(H, loc, \tau) \end{array}}{<K,H,SF::<own, m, pc, V, S::loc>> \ \Rightarrow \ <K,H,SF::<own, m, pc+1, V, S::loc>>} \quad \text{(R 19.1)}$$

Verification:

$$\boxed{\begin{array}{l} \tau \in ReferenceType \\ checkType(loc, \texttt{r}) \end{array}}$$

Runtime Exceptions:

$$\dfrac{\begin{array}{l} I = \texttt{checkcast } \tau \\ exception = \left\{\begin{array}{ll} \texttt{SecurityException} & \text{if } loc \neq \texttt{null} \ \wedge \ \neg checkCast(own.context, \tau, H(loc)) \\ \texttt{ClassCastException} & \text{if } \neg checkCanCast(H, loc, \tau) \end{array}\right. \\ loc' = getExceptionLocation(exception) \\ SF' = catchException(SF::<own, m, pc, V, S::loc>, H, loc') \end{array}}{<K,H,SF::<own, m, pc, V, S::loc>> \ \Rightarrow \ <K,H,SF'>} \quad \text{(R 19.2)}$$

If the object reference *loc* cannot be cast to the specified array type or the resolved class or interface type, the `checkcast` instruction throws a `ClassCastException`.

InstanceOfInstruction

Instruction	`instanceof` τ
Short Description	Determine if object is of reference type τ
JCVM bytecodes	*instanceof*

$$\frac{\begin{array}{l} I = \texttt{instanceof}\ \tau \\ v = \begin{cases} 1 & loc \neq \texttt{null} \ \wedge\ H \vdash loc\ :\preceq\ \tau \\ 0 & \text{otherwise} \end{cases} \\ loc \neq \texttt{null} \ \Rightarrow\ checkCast(own.context, \tau, H(loc)) \end{array}}{<K,H,SF::<own,\ m, pc,\ V,\ S::loc>> \ \Rightarrow\ <K,H,SF::<own,\ m, pc{+}1,\ V,\ S::v^{\mathsf{s}}>>} \quad \text{(R 20.1)}$$

Verification:

$$\boxed{\begin{array}{l} \tau \in ReferenceType \\ checkType(loc, \texttt{r}) \end{array}}$$

Runtime Exceptions:

$$\frac{\begin{array}{l} I = \texttt{instanceof}\ \tau \\ loc \neq \texttt{null} \ \wedge\ \neg checkCast(own.context, \tau, H(loc)) \\ exception = \texttt{SecurityException} \\ loc' = getExceptionLocation(exception) \\ SF' = catchException(SF::<own,\ m, pc,\ V,\ S::loc>, H, loc') \end{array}}{<K,H,SF::<own,\ m, pc,\ V,\ S::loc>> \ \Rightarrow\ <K,H,SF'>} \quad \text{(R 20.2)}$$

9.3 Method Support [Mx]

InvokeDefiniteMethodInstruction

Instruction	`invokedefinite` m
Short Description	Invoke definite (static or special) method m
JCVM bytecodes	*invokestatic*, *invokespecial*

The `invokedefinite` instruction contains a resolved method as argument. No method search through the class hierarchy is neccesary, even if the original call was made to an inherited method. The method is resolved statically i.e. before execution. The information on the original (subclass) class where the method was invoked is not present.

- Static Methods

$$\frac{\begin{array}{l} I = \texttt{invokedefinite}\ m' \ \ \wedge\ \ m'.isStatic \\ (\tau_i)_1^n \rightarrow \tau_r = m'.type \\ V' = v_1 :: \ldots :: v_n \ \wedge\ F' = <own,\ m',0,\ V',\epsilon> \end{array}}{<K,H,SF::<own,\ m, pc,\ V,\ S::v_1::\ldots::v_n>> \ \Rightarrow\ <K,H,SF::<own,\ m, pc,\ V,\ S>::F'>} \quad \text{(R 21.1)}$$

Verification:

$$\boxed{\begin{array}{l} \neg m'.\mathit{isInit} \\ \forall\, 0 < i \leq n.\;\; H \vdash v_i \;:\preceq\; \tau_i \\ m'.\mathit{nbArgumentWords} = \mathit{nbWords}(v_1{::}\ldots{::}v_n) \end{array}}$$

- Non-static methods:

$$\frac{\begin{array}{l} I = \texttt{invokedefinite}\;\; m' \;\;\wedge\;\; \neg m'.\mathit{isStatic} \\ \mathit{checkNull}(\mathit{loc}) \;\wedge\; o = H(\mathit{loc}) \\ (\tau_i)_1^n \rightarrow \tau_r = m'.\mathit{type} \\ \mathit{checkInvokeDefinite}(\mathit{own.context}, o) \\ V' = \mathit{loc} :: v_1 :: \ldots :: v_n \;\wedge\; F' = <o.\text{owner}, m', 0, V', \epsilon> \end{array}}{<K,H,SF::<\mathit{own}, m, \mathit{pc}, V, S::\mathit{loc}::v_1::\ldots::v_n>> \;\Rightarrow\; <K,H,SF::<\mathit{own}, m, \mathit{pc}, V, S>::F'>} \quad \text{(R 21.2)}$$

Verification:

$$\boxed{\begin{array}{l} \mathit{checkType}(\mathit{loc}, \texttt{r}) \\ (m'.\mathit{isInit} \;\Rightarrow\; \neg o.\mathit{isInit}) \;\wedge\; (\neg m'.\mathit{isInit} \;\Rightarrow\; o.\mathit{isInit}) \\ \forall\, 0 < i \leq n.\;\; H \vdash v_i \;:\preceq\; \tau_i \\ m'.\mathit{nbArgumentWords} = \mathit{nbWords}(\mathit{loc}::v_1::\ldots::v_n) \\ \left(\begin{array}{l} m'.\mathit{isProtected} \wedge \\ m'.\mathit{classI} \in \mathit{superClass}^*(m.\mathit{classI}) \wedge \\ m'.\mathit{classI.package} \neq m.\mathit{classI.package} \end{array}\right) \Rightarrow o.\mathit{class} \preceq m.\mathit{classI} \end{array}}$$

Runtime Exception:

$$\frac{\begin{array}{l} I = \texttt{invokedefinite}\;\; m' \;\;\wedge\;\; \neg m'.\mathit{isStatic} \\ (\tau_i)_1^n \rightarrow \tau_r = m'.\mathit{type} \\ \mathit{exception} = \left\{\begin{array}{ll} \texttt{NullPointerException} & \text{if } \neg\mathit{checkNull}(\mathit{loc}) \\ \texttt{SecurityException} & \text{if } \neg\mathit{checkInvokeDefinite}(\mathit{own.context}, H(\mathit{loc})) \end{array}\right. \\ \mathit{loc}' = \mathit{getExceptionLocation}(\mathit{exception}) \\ \mathit{SF}' = \mathit{catchException}(SF::<\mathit{own}, m, \mathit{pc}, V, S::\mathit{loc}::v_1::\ldots::v_n>, H, \mathit{loc}') \end{array}}{<K,H,SF::<\mathit{own}, m, \mathit{pc}, V, S::\mathit{loc}::v_1::\ldots::v_n>> \;\Rightarrow\; <K,H,SF'>} \quad \text{(R 21.3)}$$

InvokeVirtualMethodInstruction

Instruction	`invokevirtual` m
Short Description	Invoke virtual method m
JCVM bytecodes	*invokevirtual*

$$\frac{\begin{array}{l} I = \texttt{invokevirtual}\;\; m' \\ \mathit{checkNull}(\mathit{loc}) \;\wedge\; o = H(\mathit{loc}) \\ (\tau_i)_1^n \rightarrow \tau_r = m'.\mathit{type} \\ m_v = \left\{\begin{array}{ll} \mathit{methodLookup}(m'.\mathit{id}, \texttt{Object}) & o \in \mathit{ArrayObject} \\ \mathit{methodLookup}(m'.\mathit{id}, o.\mathit{class}) & \text{otherwise} \end{array}\right. \\ \mathit{appCtxt} = \mathit{getAppletContext}(SF::<\mathit{own}, m, \mathit{pc}, V, S::\mathit{loc}::v_1::\ldots::v_n>) \\ \mathit{checkInvokeVirtual}(\mathit{own.context}, \mathit{appCtxt}, o) \\ V' = \mathit{loc} :: v_1 :: \ldots :: v_n \;\wedge\; F' = <o.\mathit{owner}, m_v, 0, V', \epsilon> \end{array}}{<K,H,SF::<\mathit{own}, m, \mathit{pc}, V, S::\mathit{loc}::v_1::\ldots::v_n>> \;\Rightarrow\; <K,H,SF::<\mathit{own}, m, \mathit{pc}, V, S>::F'>} \quad \text{(R 22.1)}$$

Verification:

$$\begin{array}{|l|}\hline checkType(loc, \mathtt{r}) \\ \neg m'.isStatic \;\wedge\; \neg m'.isInit \\ o.isInit \\ \forall\, 0 < i \leq n.\;\; H \vdash v_i \;\; :\preceq \;\tau_i \\ m'.nbArgumentWords = nbWords(loc::v_1::\ldots::v_n) \\ \left(\begin{array}{l} m'.isProtected \,\wedge \\ m'.classI \in superClass(m.classI) \,\wedge \\ m'.classI.package \neq m.classI.package \end{array}\right) \Rightarrow o.class \preceq m.classI \\ \hline \end{array}$$

Runtime Exceptions:

(R 22.2)

$$\frac{\begin{array}{l} I = \mathtt{invokevirtual}\;\; m' \\ (\tau_i)_1^n \rightarrow \tau_r = m'.type \\ appCtxt = getAppletContext(SF::<own, m, pc, V, S::loc::v_1::\ldots::v_n>) \\ exception = \left\{\begin{array}{ll} \mathtt{NullPointerException} & \text{if } \neg checkNull(loc) \\ \mathtt{SecurityException} & \text{if } \neg checkInvokeVirtual(own.context, appCtxt, H(loc)) \end{array}\right. \\ loc' = getExceptionLocation(exception) \\ SF' = catchException(SF::<own, m, pc, V, S::loc::v_1::\ldots::v_n>, H, loc') \end{array}}{<K,H,SF::<own, m, pc, V, S::loc::v_1::\ldots::v_n>> \;\Rightarrow\; <K,H,SF'>}$$

InvokeInterfaceMethodInstruction

Instruction	`invokeinterface` m
Short Description	Invoke interface method m
JCVM bytecodes	*invokeinterface*

$$\frac{\begin{array}{l} \mathtt{invokeinterface}\;\; m' \\ checkNull(loc) \;\wedge\; o = H(loc) \\ (\tau_i)_1^n \rightarrow \tau_r = m'.type \\ m_v = \left\{\begin{array}{ll} methodLookup(m'.id, \mathtt{Object}) & o \in ArrayObject \\ methodLookup(m'.id, o.class) & \text{otherwise} \end{array}\right. \\ appCtxt = getAppletContext(SF::<own, m, pc, V, S::loc::v_1::\ldots::v_n>) \\ checkInvokeIface(own.context, appCtxt, m'.classI, o) \\ V' = loc :: v_1 :: \ldots :: v_n \;\wedge\; F' = <o.owner, m_v, 0, V', \epsilon> \end{array}}{<K,H,SF::<own, m, pc, V, S::loc::v_1::\ldots::v_n>> \;\Rightarrow\; <K,H,SF::<own, m, pc, V, S>::F'>} \quad \text{(R 23.1)}$$

Verification:

$$\begin{array}{|l|}\hline iface = m'.classI \in Interface \\ \neg m'.isInit \;\wedge\; m'.isPublic \\ checkType(loc, \mathtt{r}) \;\wedge\; o.isInit \\ iface \in implements(o.class) \\ \forall\, 0 < i \leq n.\;\; H \vdash v_i \;\; :\preceq \;\tau_i \\ m'.nbArgumentWords = nbWords(loc::v_1::\ldots::v_n) \\ \hline \end{array}$$

Runtime exceptions:

$$\frac{\begin{array}{l} I = \texttt{invokeinterface}\ m' \\ (\tau_i)_1^n \rightarrow \tau_r = m'.type \\ ctxt = own.context \wedge appCtxt = getAppletContext(SF{::}{<}own, m, pc, V, S{::}loc{::}v_1{::}\ldots{::}v_n{>}) \\ exception = \begin{cases} \texttt{NullPointerException} & \text{if } \neg checkNull(loc) \\ \texttt{SecurityException} & \text{if } \neg checkInvokeIface(ctxt, appCtxt, m'.classI, H(loc)) \end{cases} \\ loc' = getExceptionLocation(exception) \\ SF' = catchException(SF{::}{<}own, m, pc, V, S{::}loc{::}v_1{::}\ldots{::}v_n{>}, H, loc') \end{array}}{{<}K,H,SF{::}{<}own, m, pc, V, S{::}loc{::}v_1{::}\ldots{::}v_n{>}{>} \Rightarrow {<}K,H,SF'{>}} \quad \text{(R 23.2)}$$

ReturnInstruction

Instruction	`return` t_{opt}
Short Description	Return `void` or value type t from method
JCVM bytecodes	*return, areturn, sreturn*

$$\frac{\begin{array}{c} I = \texttt{return} \\ F' = {<}own', m', pc', V', S'{>} \end{array}}{{<}K,H,SF{::}F'{::}{<}own, m, pc, V, S{>}{>} \Rightarrow {<}K,H,SF{::}{<}own', m', pc'{+}1, V', S'{>}{>}} \quad \text{(R 24.1)}$$

$$\frac{I = \texttt{return}}{{<}K,H,{<}own, m, pc, V, S{>}{>} \Rightarrow {<}K,H,\texttt{halt}{>}} \quad \text{(R 24.2)}$$

$$\frac{\begin{array}{c} I = \texttt{return}\ t \\ F' = {<}own', m', pc', V', S'{>} \end{array}}{{<}K,H,SF{::}F'{::}{<}own, m, pc, V, S{::}v{>}{>} \Rightarrow {<}K,H,SF{::}{<}own', m', pc'{+}1, V', S'{::}v{>}{>}} \quad \text{(R 24.3)}$$

Verification: $\boxed{checkType(v, t)}$

Description: Any values on the operand stack are discarded. The virtual machine then reinstates the frame of the invoker and returns control to the invoker.

9.4 The Array Language [A1]

ArrayLengthInstruction

Instruction	`arraylength`
Short Description	Get length of array
JCVM bytecodes	*arraylength*

$$\frac{\begin{array}{l} I = \texttt{arraylength} \\ checkNull(loc) \wedge v = H(loc).length \\ appCtxt = getAppletContext(SF{::}{<}own, m, pc, V, S{::}loc{>}) \\ checkArrayLoad(own.context, appCtxt, H(loc)) \end{array}}{{<}K,H,SF{::}{<}own, m, pc, V, S{::}loc{>}{>} \Rightarrow {<}K,H,SF{::}{<}own, m, pc{+}\ 1, V, S{::}v^{s}{>}{>}} \quad \text{(R 25.1)}$$

Verification: $\boxed{H \vdash loc\ :\ \tau\ \wedge\ \tau \in ArrayType}$

Runtime Exceptions:

$$\frac{\begin{array}{l} I = \mathtt{arraylength} \\ appCtxt = getAppletContext(SF{::}{<}own, m, pc, V, S{::}loc{>}) \\ exception = \left\{ \begin{array}{ll} \mathtt{NullPointerException} & \text{if } \neg checkNull(loc) \\ \mathtt{SecurityException} & \text{if } \neg checkArrayLoad(own.context, appCtxt, H(loc)) \end{array} \right. \\ loc' = getExceptionLocation(exception) \\ SF' = catchException(SF{::}{<}own, m, pc, V, S{::}loc{>}, H, loc') \end{array}}{{<}K, H, SF{::}{<}own, m, pc, V, S{::}loc{>}{>} \Rightarrow {<}K, H, SF'{>}} \quad \text{(R 25.2)}$$

ArrayLoadInstruction

Instruction	`arrayload` t
Short Description	Load value from array of type t
JCVM bytecodes	$\{a,b,i,s\}$*aload*

$$\frac{\begin{array}{l} I = \mathtt{arrayload}\ t \\ checkNull(loc)\ \wedge\ checkArrayBounds(H(loc), i) \\ v = \left\{ \begin{array}{ll} toShort(H(loc).values(i)) & \text{if } t = \mathtt{b} \\ H(loc).values(i) & \text{otherwise} \end{array} \right. \\ appCtxt = getAppletContext(SF{::}{<}own, m, pc, V, S{::}loc{::}i{>}) \\ checkArrayLoad(own.context, appCtxt, H(loc)) \end{array}}{{<}K, H, SF{::}{<}own, m, pc, V, S{::}loc{::}i{>}{>} \Rightarrow {<}K, H, SF{::}{<}own, m, pc{+}\ 1, V, S{::}v{>}{>}} \quad \text{(R 26.1)}$$

Verification: $\boxed{\begin{array}{l} checkType(loc, \mathtt{r})\ \ \wedge\ \ checkType(i, \mathtt{s}) \\ H \vdash loc\ :\ (\mathtt{array}\ \tau_e) \end{array}}$

Runtime exceptions:

$$\frac{\begin{array}{l} I = \mathtt{arrayload}\ t \\ appCtxt = getAppletContext(SF{::}{<}own, m, pc, V, S{::}loc{::}i{>}) \\ exception = \left\{ \begin{array}{ll} \mathtt{NullPointerException} & \text{if } \neg checkNull(loc) \\ \mathtt{SecurityException} & \text{if } \neg checkArrayLoad(own.context, appCtxt, H(loc)) \\ \mathtt{ArrayIndexOutOfBoundsException} & \text{if } \neg checkArrayBounds(H(loc), i) \end{array} \right. \\ loc' = getExceptionLocation(exception) \\ SF' = catchException(SF{::}{<}own, m, pc, V, S{::}loc{::}i{>}, H, loc') \end{array}}{{<}K, H, SF{::}{<}own, m, pc, V, S{::}loc{::}i{>}{>} \Rightarrow {<}K, H, SF'{>}} \quad \text{(R 26.2)}$$

ArrayStoreInstruction

Instruction	`arraystore` t
Short Description	Store value into array of type t
JCVM bytecodes	$\{a,b,i,s\}$*astore*

$$\frac{\begin{array}{l} I = \texttt{arraystore}\ t \\ \mathit{checkNull}(loc)\ \wedge\ \mathit{checkArrayBounds}(H(loc), i) \\ \mathit{values}' = (H(loc).\mathit{values})[i \mapsto v]\ \wedge\ o' = H(loc)[\mathit{values} \mapsto \mathit{values}'] \\ v' = \left\{ \begin{array}{ll} \mathit{fromShort}(v, \texttt{b}) & \text{if } t = \texttt{b} \\ v & \text{otherwise} \end{array} \right. \ \wedge\ H' = H[loc \mapsto o'] \\ \mathit{appCtxt} = \mathit{getAppletContext}(SF{::}{<}own, m, pc, V, S{::}loc{::}i{::}v{>}) \\ \mathit{checkArrayStore}(own.context, \mathit{appCtxt}, H(loc), H, v) \end{array}}{<K,H,SF{::}{<}own, m, pc, V, S{::}loc{::}i{::}v{>}{>}\ \Rightarrow\ <K,H',SF{::}{<}own, m, pc+1, V, S{>}{>}} \quad \text{(R 27.1)}$$

Verification:

$\mathit{checkType}(loc, \texttt{r})\ \wedge\ \mathit{checkType}(i, \texttt{s})$
$\mathit{checkType}(v, t)$
$H \vdash loc\ :\ (\texttt{array}\ \tau_e)$
$H \vdash v\ :\preceq\ \tau_e$

Runtime exceptions:

$$\frac{\begin{array}{l} I = \texttt{arraystore}\ t \\ loc \neq \texttt{null}\ \wedge\ 0 \leq i < H(loc).length \\ \mathit{appCtxt} = \mathit{getAppletContext}(SF{::}{<}own, m, pc, V, S{::}loc{::}i{::}v{>}) \\ \mathit{exception} = \left\{ \begin{array}{ll} \texttt{NullPointerException} & \text{if } \neg\mathit{checkNull}(loc) \\ \texttt{SecurityException} & \text{if } \neg\mathit{checkArrayStore}(own.context, H(loc), H, v) \\ \texttt{ArrayIndexOutOfBoundsException} & \text{if } \neg\mathit{checkArrayBounds}(H(loc), i) \end{array} \right. \\ loc' = \mathit{getExceptionLocation}(\mathit{exception}) \\ SF' = \mathit{catchException}(SF{::}{<}own, m, pc, V, S{::}loc{::}i{::}v{>}, H, loc') \end{array}}{<K,H,SF{::}{<}own, m, pc, V, S{::}loc{::}i{::}v{>}{>}\ \Rightarrow\ <K,H,SF'{>}} \quad \text{(R 27.2)}$$

9.5 The Exception support [E3]

ThrowInstruction

Instruction	`throw`
Short Description	Throw exception or error
JCVM bytecodes	*athrow*

Description: The `throw` instruction throws the exception (object) referenced by the element on top of the stack.

$$\frac{\begin{array}{l} I = \texttt{throw} \\ H \vdash loc\ :\preceq\ \texttt{Throwable} \\ \mathit{checkNull}(loc)\ \wedge\ \mathit{checkThrow}(own.context, H(loc)) \\ SF' = \mathit{catchException}(SF :: {<}own, m, pc, V, S{>}, H, loc) \end{array}}{<K,H,SF{::}{<}own, m, pc, V, S{::}loc{>}{>}\ \Rightarrow\ <K,H,SF'{>}} \quad \text{(R 28.1)}$$

The location *loc* on top of the stack must be a reference to an object that is an instance of class `Throwable` or of a subclass of `Throwable`. The process of catching the exception referenced by *loc* is performed by $\mathit{catchException}(SF :: {<}own, m, pc, V, S{>}, H, loc)$, where *loc* has been popped from the operand stack of the top frame. The semantics of *catchException* is described in section 6.

Runtime Exceptions:

$$\frac{\begin{array}{l} I = \texttt{throw} \\ exception = \begin{cases} \texttt{NullPointerException} & \text{if } \neg checkNull(loc) \\ \texttt{SecurityException} & \text{if } \neg checkThrow(own.context, H(loc)) \end{cases} \\ loc' = getExceptionLocation(exception) \\ SF' = catchException(SF{::}{<}own,\, m, pc,\, V,\, S{::}loc{>}, H, loc') \end{array}}{<K,H,SF{::}{<}own,\, m, pc,\, V,\, S{::}loc{>}{>} \;\Rightarrow\; <K,H,SF'>} \quad \text{(R 28.2)}$$

9.6 The Subroutine Support [S]

JumpSubroutineInstruction

Instruction	`jsr` *addr*
Short Description	Jump to subroutine at address *addr*
JCVM bytecodes	*jsr*

$$\frac{I = \texttt{jsr}\;\; addr}{<K,H,SF{::}{<}own,\, m, pc,\, V,\, S{>}{>} \;\Rightarrow\; <K,H,SF{::}{<}own,\, m,\, addr^{\texttt{ra}},\, V,\, S{::}pc{+}1{>}{>}} \quad \text{(R 29.1)}$$

Verification: $\boxed{method.instructionAt(addr) \neq \bot}$

ReturnSubroutineInstruction

Instruction	`ret` *i*
Short Description	Return from subroutine to address in local variable *i*
JCVM bytecodes	*ret*

$$\frac{I = \texttt{ret}\;\; i \;\wedge\;\; \wedge\, pc' = V[i]}{<K,H,SF{::}{<}own,\, m, pc,\, V,\, S{>}{>} \;\Rightarrow\; <K,H,SF{::}{<}own,\, m, pc',\, V,\, S{>}{>}} \quad \text{(R 30.1)}$$

Verification: $\boxed{checkType(V[i], \texttt{ra})}$

9.7 The Integer and Multi-Word Operations [I]

Already considered by the less-specialised instruction set.

References

[1] Marc Eluard and Thomas Jensen. Towards an operational semantics for Java Card bytecode. Technical Report SECSAFE-IRISA-001, v1.0, IRISA, May 2001. SecSafe confidential.

[2] Rene Rydhof Hansen. Flow logic for JVML. Technical Report SECSAFE-DAIMI-005, v2.1, DAIMI, February 2001. SecSafe Internal.

[3] Renaud Marlet. Proposition of a hierarchy of languages to be studied in SecSafe. Technical Report SECSAFE-TL-004, v1.0, Trusted Logic, December 2000. SecSafe public.

[4] Renaud Marlet. Syntax of the jcvm language to be studied in the secsafe project. Technical Report SECSAFE-TL-005, v1.7, Trusted Logic, May 2001. SecSafe public, http://www.doc.ic.ac.uk/ siveroni/secsafe.

[5] Sun Microsystems, Inc., Palo Alto, CA 94303 USA. *The Java Card^{TM} 2.1.1 Runtime Environemnt (JCRE) Specification*, revision 1.0 edition, May 18 200.

[6] Sun Microsystems, Inc., Palo Alto, CA 94303 USA. *The Java Card^{TM} 2.1.1 Virtual Machine Specification*, revision 1.0 edition, May 18 2000.