The operational semantics of a Java Secure Processor

Pieter H. Hartel, Michael J. Butler and Moshe Levy
{phh,mjb}@ecs.soton.ac.uk, Moshe.Levy@Sun.COM

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-98-1
January 16 1998

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ, United Kingdom

The operational semantics of a Java Secure Processor

Pieter H. Hartel[1], Michael J. Butler[1], Moshe Levy[2]

[1] Dept. of Electronics and Computer Science, Univ. of Southampton, UK, Email: {phh,mjb}@ecs.soton.ac.uk
[2] JavaSoft, Inc. A Sun Microsystems, Inc. Business, Palo Alto, CA 94043 USA Email: Moshe.Levy@Sun.COM

Abstract. A formal specification of a Java Secure Processor is presented, which is mechanically checked for type consistency, well formedness and operational conservativity. The specification is executable and it is used to animate and study the behaviour of sample Java programs. The purpose of the semantics is to document the behaviour of the complete JSP for the benefit of implementors.

1 Introduction

A smart card is a complete 'embedded' computer housed in a piece of plastic the same size as a credit card [12]. The computer has to be small to reduce the risk of mechanical problems. Because of these mechanical constraints, as well as aspects of cost, the current generation of smart cards typically contains only a small 8-bit micro processor, a few hundred bytes of RAM, a few Kbytes of ROM and a few Kbytes of EEPROM. This small size constrains the freedom in the design of the software that has to be run on a smart card processor.

Java [4] was originally designed for writing embedded software. Because of this pedigree it is attractive to be used as a smart card programming language. Some facilities provided by the Java language are too expensive to be implemented on a smart card. Threads, and dynamic class file loading fall in this category. Further study is needed to find ways of incorporating the Java exception mechanism and a garbage collector on smart cards. Smart cards do not use floating point arithmetic so this feature of Java is not needed. Using the subset of Java as described above for smart cards is attractive. It is also feasible to implement this Java subset on computers with limited resources.

The standard Java class libraries are not suitable for smart cards because many of the facilities provided are meaningless on smart cards. Examples include the interface to GUI libraries. Instead a smart card would host a specially designed set of class libraries dedicated to the application domain of card applications. The set of class libraries would be small enough to fit in the card and would be versatile enough to provide standard smart card facilities, such as the ISO 7816-4 command set [1], or down loadable applications for multi-application smart cards [9].

A Java Secure Processor (JSP) is a virtual machine that is designed to fit on a smart card. A JSP does not implement the full Java Virtual Machine (JVM) [7]. Instead a JSP is accompanied by a JVM to JSP translator, which compiles standard JVM byte codes into byte codes for the JSP. Java Soft have written a sophisticated translator, which performs extensive program analysis to allow a large class of Java programs to be run on the JSP. To support our work on the formal definition of the operational semantics of the JSP we have written a simple translator, which accepts rather a smaller class of Java programs. The simple translator is used to validate the operational semantics.

A standard Java development environment can be used to write Java programs for smart cards. Instead of relying on the standard class libraries the programmer uses the smart card class libraries. A simulator can be used to test the code. The process of loading Java programs into a card is quite different from loading and running programs on a workstation, as it may involve manufacturing ROM masks. We will not discuss this aspect further, the interested reader is referred to the literature [12].

A smart card is a secure token that may control commodities of real value. Secure here means that the card should be hardware and software tamper resistant, and that it should not leak information. The considerations that apply to the security of Java in general [8] also apply to Java for smart cards. In addition Java for smart cards should provide facilities such as ownership control and cryptographically protected modes of use.

The resource limitation of a smart card makes it more difficult to ensure that security is maintained. For example currently a complete byte code verifier is too large to be implemented on a smart card. The JSP approach assumes that JVM byte codes are verified when translated into JSP byte codes. The results are then digitally signed so that tampering can be detected when code is being loaded.

A clear, concise and complete specification of the semantics is a prerequisite for a successful and secure implementation of a JSP. The present document provides such a specification. The document is based on an informal description of the JSP from Java Soft, who are currently building a tool suite for a JSP [6]. The formal specification is self contained but does not document the motivation for many of the design decisions made for the JSP. The interested reader is referred to the informal specification.

The present formal specification is a `latos` [5] literate script. `Latos` is a tool for developing operational semantics. `Latos` supports publication quality rendering using LaTeX, execution and animation using a functional programming language, and derivation tree browsing using Netscape. `Latos` helps to check that a specification is operationally conservative. The `latos` meta language is basically Miranda[1] [11] augmented with a notation for rules of inference and sets. Developing a semantics as a literate script avoids clerical errors and confusion, as syntax and type errors are detected by the tool.

The formal specification does not support the capabilities of JSP development

[1] Miranda is a trademark of Research Software Ltd.

environment. Instead the `latos` tool provides a tracing facility allowing for a detailed study and analysis of executing application programs.

Related work on the semantics of the JVM includes the executable specification of the 'defensive' JVM made by Computational Logic Inc [3], and work by Bertelson on another subset of the JVM [2].

The next section describes the restrictions imposed on the kind of Java programs supported by a JSP on a smart card. Section 3 presents the execution model of a JSP and Section 4 defines the instructions of the virtual machine. The relationship between the JVM and the JSP is explored in Section 5. A brief example of how the semantics of the JSP may be used to validate the behaviour of a sample Java program is given in Section 6. The last section presents our conclusions and suggestions for future work.

2 Java language restrictions

The JSP design imposes a number of restrictions to allow a Java program to be run in the constrained runtime environment of a smart card. The most important restrictions are:

- The JSP provides no support threads, multi-dimensional arrays, floating point numbers, and Just-in-time byte code translation.
- Exceptions may be raised by application programs, but they can only be handled by the system.
- There is no garbage collection. Objects can be allocated dynamically but the majority are expected to be allocated statically using compile time garbage collection techniques. The formal specification allows objects to be allocated any time. It would be possible to state and prove a property about programs that are guaranteed not to allocate objects after a certain point in their execution. This constitutes a desirable safety property of those programs.
- Class files cannot be loaded dynamically. Instead the software to be present in the card is loaded when the card is manufactured or personalised.
- Recursive methods are discouraged and recursive class constructors and exceptions are disallowed.
- Integers and shorts are identified. The JVM to JSP byte code translator should ensure that the results obtained from a computation on the JSP are identical to the results that would have been obtained on a JVM.
- The number of arguments, local variables, methods, and object instances are limited.

Java programmers have to be aware of these restrictions when writing code that is intended for a JSP. Some of the restrictions can be circumvented by the use of appropriate class libraries. Others will be taken care of by program analysis techniques in the JVM to JSP translator.

3 Execution model

The JSP is a byte oriented stack machine. It also has a read-only memory area for storing methods and constants, an area of memory and some registers to maintain the book-keeping of the machine, and a heap.

The data manipulated directly by Java programs is faithfully modelled by the semantics. In particular the operand stack, the fields of objects and the elements of arrays contain bytes only. A short or a reference is always treated as a pair of bytes. The structures that support the machine itself, such as the byte codes, stack frames and heap objects are modelled as higher level entities rather than as collections of bytes. The ensuing specification is of a low level, which makes it eminently suitable to serve as a guideline for implementors of a JSP.

The formal specification defines all structured data (not scalars) of the virtual machine either as mappings or as algebraic data types (i.e. a sum of product types). Each of these is of a different type, that is incompatible with any other useful type. The `latos` system performs strong type checking to ensure that all the type constraints in the operational semantics are indeed satisfied.

3.1 Basic data

The basic data in the formal specification are derived from the natural numbers. Similarly, the raw data in the JSP implementation are derived from a sequence of bytes. The type bit (below) permits any numeric value, but sensible values are in the range $0 \ldots 1$. (The equivalence symbol is used to bind a name to a type, the equals symbol binds a name to a value). In a JSP implementation, a boolean is stored in a byte, which permits sensible values as well as non-sensible values. We would have preferred to identify bit and $\mathsf{bit}_{\mathsf{range}}$ but unfortunately the type system used by `latos` (i.e. the Hindley-Milner type system of Miranda) is not strong enough to support sub types.

$\mathsf{bit} \quad \equiv \mathsf{num};$

$\mathsf{bit}_{\mathsf{range}} = 0 \ \ldots \ 1;$

Other raw data and ranges defined in a similar way include the signed 8-bit byte, the signed 16-bit short and the unsigned 16-bit reference. The nullreference is a special reference value, which is represented as zero. Regular references should not have this particular value.

3.2 Store areas

The JSP virtual machine uses a number of areas of store for data, code and book-keeping. Each of these areas is represented in the formal specification as a mapping of numerical indices onto values of the appropriate type, thus providing a uniform, albeit low level approach to information handling in the JSP.

- A JSP uses a stack of activation frames, where each frame contains an operand stack and some bookkeeping. The activation frames are gathered in the machine-wide $\mathsf{frameArea}$. The frame area is represented as a partial mapping from the domain $\mathsf{framePointer}$ to the range frame. The representation as a partial function makes it possible to represent common operations on structures in a clear and succinct way. The type frame itself is defined in Section 3.3.

$\mathsf{framePointer} \equiv \mathsf{num};$
$\mathsf{framePointer}_{\mathsf{range}} = 0 \ldots 255;$
$\mathsf{frameArea} \equiv \mathsf{framePointer} \not\rightarrow \mathsf{frame};$

– Heap objects are instances of classes or arrays. The objects are gathered in the machine-wide $\mathsf{heapArea}$. The type object is defined in Section 3.5.

$\mathsf{heapPointer} \equiv \mathsf{num};$
$\mathsf{heapPointer}_{\mathsf{range}} = 0 \ldots 65535;$
$\mathsf{heapArea} \equiv \mathsf{heapPointer} \not\rightarrow \mathsf{object};$

– Static program data are represented by bytes. This data is gathered in the machine-wide $\mathsf{staticArea}$.

$\mathsf{staticPointer} \equiv \mathsf{num};$
$\mathsf{staticPointer}_{\mathsf{range}} = 0 \ldots 65535;$
$\mathsf{staticArea} \equiv \mathsf{staticPointer} \not\rightarrow \mathsf{byte};$

– The machine-wide $\mathsf{codeArea}$ gathers the byte codes and the method headers for the methods of all application programs in the system. The type $\mathsf{byteCode}$ is defined in Section 4.

$\mathsf{programCounter} \equiv \mathsf{num};$
$\mathsf{programCounter}_{\mathsf{range}} = 0 \ldots 65535;$
$\mathsf{codeArea} \equiv \mathsf{programCounter} \not\rightarrow \mathsf{byteCode};$

– The application program table $\mathsf{progTable}$ records the class table of each loaded application program.

$\mathsf{progId} \equiv \mathsf{num};$
$\mathsf{progId}_{\mathsf{range}} = 0 \ldots 63;$
$\mathsf{progTable} \equiv \mathsf{progId} \not\rightarrow \mathsf{classTable};$

– There is one instance of class Class for each class in the system. The class table gathers such instances. The type $\mathsf{classObject}$ is defined in Section 3.5.

$\mathsf{classId} \equiv \mathsf{num};$
$\mathsf{classId}_{\mathsf{range}} = 0 \ldots 127;$
$\mathsf{classTable} \equiv \mathsf{classId} \not\rightarrow \mathsf{classObject};$

– Each class in the system is accompanied by a method table, which maps a method id onto the program counter value at which the method header is located. The $\mathsf{methodTable}$ is defined as an algebraic data type with two components and with constructor $\mathbf{MethodTable}$.

$\mathsf{methodId} \equiv \mathsf{num};$
$\mathsf{methodId}_{\mathsf{range}} = 0 \ldots 255;$
$\mathsf{entryTable} \equiv \mathsf{methodId} \not\rightarrow \mathsf{programCounter};$
$\mathsf{methodTable} \equiv \mathbf{MethodTable}\ \mathsf{classId}\ \mathsf{entryTable};$

3.3 Stack frames

The operand stack within the topmost frame plays a special role in that it can be accessed by the JSP instructions. To acknowledge this special role, the formal specification shadows the operand stack, and manipulates it as a separate component of the virtual machine configuration.

A method invocation creates a stack frame (shown below as an instance of the data type frame). The frame has the following four components:

- the programCounter representing the return address to the caller of the method.
- the framePointer to the previous frame. This information is redundant in the specification, as frames are numbered sequentially starting from 0. In an implementation frames would be referred to by their address, in which case the frame pointer is needed.
- the stackPointer within the operand stack;
- the operandStack, which contains parameters, locals and temporaries of the current method.

In the specifications that follow, a stack is always accompanied by a stack pointer (which points at the last used element). All stack operations can be modelled by a combination of adding (or subtracting) a constant to (from) the stack pointer and/or updating the mapping. For example, pushing an element onto the stack means incrementing the pointer and updating the mapping with a new association.

$$
\begin{array}{ll}
\mathsf{stackPointer} & \equiv \mathsf{num}; \\
\mathsf{stackPointer}_{\mathsf{range}} & = 0 \ldots 255; \\
\mathsf{operandStack} & \equiv \mathsf{stackPointer} \not\rightarrow \mathsf{byte}; \\
\mathsf{frame} & \equiv \mathbf{Frame}\ \mathsf{programCounter}\ \mathsf{framePointer} \\
 & \quad \mathsf{stackPointer}\ \mathsf{operandStack};
\end{array}
$$

A JSP uses a slightly different stack frame configuration than the JVM, a difference that is taken into account by the JVM to JSP byte code translator.

3.4 Headers

Objects and methods have headers, which record book-keeping information. This section describes all possible headers in the system.

- An objectHeader records the identity of the application program progId, the size of the object in bytes instanceSize and a table listing all the methods for the object. The classId for the object is available from the methodTable.

$$
\begin{array}{ll}
\mathsf{instanceSize} & \equiv \mathsf{num}; \\
\mathsf{instanceSize}_{\mathsf{range}} & = 0 \ldots 127; \\
\mathsf{objectHeader} & \equiv \mathbf{ObjectHeader}\ \mathsf{progId}\ \mathsf{instanceSize}\ \mathsf{methodTable};
\end{array}
$$

- An array may contain scalars (bits, bytes or shorts) or references to objects. An array has a header, which records the application program id, the class id of the element type, the method table for the element type, an indication of the element type and the length of the array.

$$
\begin{array}{ll}
\mathsf{dataType} & \equiv \mathbf{bit} \mid \mathbf{byte} \mid \mathbf{short} \mid \mathbf{ref}; \\
\mathsf{arrayLength} & \equiv \mathsf{num}; \\
\mathsf{arrayLength}_{\mathsf{range}} & = 1 \ldots 4096; \\
\mathsf{arrayHeader} & \equiv \mathbf{ArrayHeader}\ \mathsf{progId}\ \mathsf{classId}\ \mathsf{methodTable} \\
 & \quad \mathsf{dataType}\ \mathsf{arrayLength};
\end{array}
$$

– A method has a header, which records two flags and three sizes. The flags record whether the method is native and whether it is public. The stack size is currently unused, but the paramsSize and localsSize are used to create appropriate frames.

$$
\begin{array}{ll}
\mathsf{isNative} & \equiv \mathsf{bool}; \\
\mathsf{isPublic} & \equiv \mathsf{bool}; \\
\mathsf{stackSize} & \equiv \mathsf{num}; \\
\mathsf{stackSize}_{\mathsf{range}} & = 0 \ldots 15; \\
\mathsf{paramsSize} & \equiv \mathsf{num}; \\
\mathsf{paramsSize}_{\mathsf{range}} & = 0 \ldots 15; \\
\mathsf{localsSize} & \equiv \mathsf{num}; \\
\mathsf{localsSize}_{\mathsf{range}} & = 0 \ldots 15; \\
\mathsf{methodHeader} & \equiv \textbf{\textsf{MethodHeader}}\ \mathsf{isNative}\ \mathsf{isPublic} \\
 & \quad \mathsf{stackSize}\ \mathsf{paramsSize}\ \mathsf{localsSize};
\end{array}
$$

3.5 Objects

The JSP works with three different kinds of objects:

– A regular object has a header and a number of fields represented by the fieldTable. The fields are represented as bytes and the methods are available from the header.

$$
\begin{array}{ll}
\mathsf{fieldId} & \equiv \mathsf{num}; \\
\mathsf{fieldId}_{\mathsf{range}} & = 0 \ldots 255; \\
\mathsf{fieldTable} & \equiv \mathsf{fieldId} \not\rightarrow \mathsf{byte}; \\
\mathsf{regularObject} & \equiv \textbf{\textsf{RegularObject}}\ \mathsf{objectHeader}\ \mathsf{fieldTable};
\end{array}
$$

– An array object records an array header as well as the array elements. The elements are represented as bytes.

$$
\begin{array}{ll}
\mathsf{arrayIndex} & \equiv \mathsf{num}; \\
\mathsf{arrayIndex}_{\mathsf{range}} & = 0 \ldots 4095; \\
\mathsf{arrayTable} & \equiv \mathsf{arrayIndex} \not\rightarrow \mathsf{byte}; \\
\mathsf{arrayObject} & \equiv \textbf{\textsf{ArrayObject}}\ \mathsf{arrayHeader}\ \mathsf{arrayTable};
\end{array}
$$

– There is one classObject for every object in the system. The classObject itself is an instance of class Class. The classObject records the normal object header as well as the size of an instance of the class, the method table for the class, the depth in the class hierarchy, the classId of the super classes and the interface classes implemented by the class. The instance size and the method table are redundant as the object header also contains this information.

$$
\begin{array}{ll}
\mathsf{classDepth} & \equiv \mathsf{num}; \\
\mathsf{classDepth}_{\mathsf{range}} & = 0 \ldots 255; \\
\mathsf{superId} & \equiv \mathsf{num}; \\
\mathsf{superId}_{\mathsf{range}} & = 0 \ldots 255; \\
\mathsf{superTable} & \equiv \mathsf{superId} \not\rightarrow \mathsf{classId}; \\
\mathsf{interfaceId} & \equiv \mathsf{num}; \\
\mathsf{interfaceId}_{\mathsf{range}} & = 0 \ldots 255; \\
\mathsf{implementTable} & \equiv \mathsf{methodId} \not\rightarrow \mathsf{methodId}; \\
\mathsf{interfaceTable} & \equiv \mathsf{interfaceId} \not\rightarrow \mathsf{implementTable}; \\
\mathsf{classObject} & \equiv \textsf{\textbf{ClassObject}}\ \mathsf{objectHeader}\ \mathsf{instanceSize}\ \mathsf{methodTable} \\
 & \quad \mathsf{classDepth}\ \mathsf{superTable}\ \mathsf{interfaceTable};
\end{array}
$$

The JSP heap is used to store regular and array objects only. A classObject is allocated statically in a area separate from the heap. The union type object therefore does not cover class objects.

$\mathsf{object} \equiv \mathsf{regularObject} \mid \mathsf{arrayObject};$

The two auxiliary predicates below are used to determine whether we are dealing with a regular object or an array object.

$$
\begin{array}{ll}
\mathsf{isRegularObject}\ \mathsf{regularObject} & = \mathsf{True}; \\
\mathsf{isRegularObject}\ \_ & = \mathsf{False}; \\
\mathsf{isArrayObject}\ \mathsf{arrayObject} & = \mathsf{True}; \\
\mathsf{isArrayObject}\ \_ & = \mathsf{False};
\end{array}
$$

4 Instruction set

There are 25 different categories of JSP byteCode (below), all with their own type. The methodHeader is treated as a pseudo instruction. This models the practice of preceding the code for each method by its header.

$$
\begin{array}{ll}
\mathsf{byteCode} \equiv & \mathsf{methodHeader} \mid \\
 & \mathsf{constInst} \mid \mathsf{loadInst} \mid \mathsf{storeInst} \mid \mathsf{incInst} \mid \mathsf{stackInst} \mid \\
 & \mathsf{newarrayInst} \mid \mathsf{arrayLoadInst} \mid \mathsf{arrayStoreInst} \mid \\
 & \mathsf{arithInst} \mid \mathsf{logicalInst} \mid \mathsf{convertInst} \mid \mathsf{compareInst} \mid \\
 & \mathsf{controlInst} \mid \mathsf{switchInst} \mid \mathsf{exceptionInst} \mid \\
 & \mathsf{invokeinterfaceInst} \mid \mathsf{invokevirtualInst} \mid \\
 & \mathsf{invokeInst} \mid \mathsf{returnInst} \mid \\
 & \mathsf{objectInst} \mid \mathsf{instanceInst} \mid \\
 & \mathsf{getfieldInst} \mid \mathsf{putfieldInst} \mid \mathsf{getstaticInst} \mid \mathsf{putstaticInst} \mid \\
 & \mathsf{breakpointInst};
\end{array}
$$

The following categories of byte codes have been defined:

- Load, store and increment instructions.

$$
\begin{aligned}
\mathsf{constInst} \equiv\ & \mathbf{nop} \mid \mathbf{bpush}\ \mathsf{byte} \mid \mathbf{spush}\ \mathsf{byte}\ \mathsf{byte} \mid \mathbf{apush}\ \mathsf{byte}\ \mathsf{byte} \mid \\
& \mathbf{aconst}_{\mathsf{null}} \mid \mathbf{bconst}_{\mathsf{m1}} \mid \\
& \mathbf{bconst}_0 \mid \mathbf{bconst}_1 \mid \mathbf{bconst}_2 \mid \mathbf{bconst}_3 \mid \mathbf{bconst}_4 \mid \mathbf{bconst}_5; \\
\mathsf{loadInst} \equiv\ & \mathbf{bload}\ \mathsf{stackPointer} \mid \mathbf{bload}_0 \mid \mathbf{bload}_1 \mid \mathbf{bload}_2 \mid \mathbf{bload}_3 \mid \\
& \mathbf{sload}\ \mathsf{stackPointer} \mid \mathbf{sload}_0 \mid \mathbf{sload}_1 \mid \mathbf{sload}_2 \mid \mathbf{sload}_3 \mid \\
& \mathbf{aload}\ \mathsf{stackPointer} \mid \mathbf{aload}_0 \mid \mathbf{aload}_1 \mid \mathbf{aload}_2 \mid \mathbf{aload}_3; \\
\mathsf{storeInst} \equiv\ & \mathbf{bstore}\ \mathsf{stackPointer} \mid \mathbf{bstore}_0 \mid \mathbf{bstore}_1 \mid \mathbf{bstore}_2 \mid \mathbf{bstore}_3 \mid \\
& \mathbf{sstore}\ \mathsf{stackPointer} \mid \mathbf{sstore}_0 \mid \mathbf{sstore}_1 \mid \mathbf{sstore}_2 \mid \mathbf{sstore}_3 \mid \\
& \mathbf{astore}\ \mathsf{stackPointer} \mid \mathbf{astore}_0 \mid \mathbf{astore}_1 \mid \mathbf{astore}_2 \mid \mathbf{astore}_3; \\
\mathsf{incInst} \equiv\ & \mathbf{binc}\ \mathsf{stackPointer}\ \mathsf{byte} \mid \mathbf{sinc}\ \mathsf{stackPointer}\ \mathsf{byte};
\end{aligned}
$$

– Stack instructions.

$$
\mathsf{stackInst} \equiv \mathbf{pop} \mid \mathbf{pop2} \mid \mathbf{dup} \mid \mathbf{dup2} \mid \mathbf{dup\_x}\ \mathsf{byte} \mid \mathbf{swap} \mid \mathbf{swap2};
$$

– Array creation, load and store instructions.

$$
\begin{aligned}
\mathsf{newarrayInst} &\equiv \mathbf{newarray}\ \mathsf{dataType} \mid \mathbf{anewarray}\ \mathsf{classId}; \\
\mathsf{arrayLoadInst} &\equiv \mathbf{arraylength} \mid \mathbf{baload} \mid \mathbf{saload} \mid \mathbf{aaload}; \\
\mathsf{arrayStoreInst} &\equiv \mathbf{bastore} \mid \mathbf{sastore} \mid \mathbf{aastore};
\end{aligned}
$$

– Instructions for arithmetical, logical, conversion and comparison operations.

$$
\begin{aligned}
\mathsf{arithInst} \equiv\ & \mathbf{bneg} \mid \mathbf{sneg} \mid \mathbf{badd} \mid \mathbf{sadd} \mid \mathbf{bsub} \mid \mathbf{ssub} \mid \\
& \mathbf{bmul} \mid \mathbf{smul} \mid \mathbf{bdiv} \mid \mathbf{sdiv} \mid \mathbf{brem} \mid \mathbf{srem}; \\
\mathsf{logicalInst} \equiv\ & \mathbf{bshl} \mid \mathbf{bshr} \mid \mathbf{bushr} \mid \mathbf{sshl} \mid \mathbf{sshr} \mid \mathbf{sushr} \mid \\
& \mathbf{band} \mid \mathbf{sand} \mid \mathbf{bor} \mid \mathbf{sor} \mid \mathbf{bxor} \mid \mathbf{sxor}; \\
\mathsf{convertInst} \equiv\ & \mathbf{s2b} \mid \mathbf{b2s}; \\
\mathsf{compareInst} \equiv\ & \mathbf{bcmp} \mid \mathbf{scmp} \mid \mathbf{acmp};
\end{aligned}
$$

– Instructions for the transfer of control.

$$
\begin{aligned}
\mathsf{offset} \equiv\ & (\mathsf{byte},\ \mathsf{byte}); \\
\mathsf{controlInst} \equiv\ & \mathbf{ifeq}\ \mathsf{offset} \mid \mathbf{iflt}\ \mathsf{offset} \mid \mathbf{ifgt}\ \mathsf{offset} \mid \\
& \mathbf{ifne}\ \mathsf{offset} \mid \mathbf{ifge}\ \mathsf{offset} \mid \mathbf{ifle}\ \mathsf{offset} \mid \mathbf{goto}\ \mathsf{offset};
\end{aligned}
$$

– Instructions to support switch statements.

$$
\begin{aligned}
\mathsf{tableswitchIndex} &\equiv \mathsf{num}; \\
\mathsf{tableswitchIndex}_{\mathsf{range}} &= 0\ \ldots\ 127; \\
\mathsf{tableswitchTable} &\equiv \mathsf{tableswitchIndex} \not\rightarrow \mathsf{offset}; \\
\mathsf{lookupswitchIndex} &\equiv \mathsf{num}; \\
\mathsf{lookupswitchIndex}_{\mathsf{range}} &= 0\ \ldots\ 126; \\
\mathsf{lookupswitchTable} &\equiv \mathsf{lookupswitchIndex} \not\rightarrow (\mathsf{byte},\ \mathsf{offset}); \\
\mathsf{switchInst} &\equiv \mathbf{tableswitch}\ \mathsf{offset}\ \mathsf{byte}\ \mathsf{byte}\ \mathsf{tableswitchTable} \mid \\
& \quad\ \mathbf{lookupswitch}\ \mathsf{offset}\ \mathsf{byte}\ \mathsf{lookupswitchTable};
\end{aligned}
$$

– Instructions to support exceptions.

$$
\mathsf{exceptionInst} \equiv \mathbf{athrow} \mid \mathbf{jsr}\ \mathsf{offset} \mid \mathbf{ret}\ \mathsf{stackPointer};
$$

– Instructions for method invokation.

$$
\begin{aligned}
\mathsf{invokeinterfaceInst} &\equiv \mathbf{invokeinterface}\ \mathsf{paramsSize}\ \mathsf{interfaceId}\ \mathsf{methodId}; \\
\mathsf{invokevirtualInst} &\equiv \mathbf{invokevirtual}\ \mathsf{paramsSize}\ \mathsf{methodId}; \\
\mathsf{invokeInst} &\equiv \mathbf{invoke}\ \mathsf{offset}; \\
\mathsf{returnInst} &\equiv \mathbf{breturn} \mid \mathbf{sreturn} \mid \mathbf{areturn} \mid \mathbf{return};
\end{aligned}
$$

– Instructions for object creation and manipulation.

$\mathsf{objectInst} \equiv \mathbf{new}\ \mathsf{classId};$
$\mathsf{instanceInst} \equiv \mathbf{instanceof}\ \mathsf{classId} \mid \mathbf{checkcast}\ \mathsf{classId} \mid$
$\quad \mathbf{ainstanceof}\ \mathsf{dataType} \mid \mathbf{acheckcast}\ \mathsf{dataType} \mid$
$\quad \mathbf{aainstanceof}\ \mathsf{classId} \mid \mathbf{aacheckcast}\ \mathsf{classId};$
$\mathsf{getfieldInst} \equiv \mathbf{bgetfield}\ \mathsf{stackPointer} \mid \mathbf{sgetfield}\ \mathsf{stackPointer};$
$\mathsf{putfieldInst} \equiv \mathbf{bputfield}\ \mathsf{stackPointer} \mid \mathbf{sputfield}\ \mathsf{stackPointer};$
$\mathsf{getstaticInst} \equiv \mathbf{bgetstatic}\ \mathsf{byte}\ \mathsf{byte} \mid \mathbf{sgetstatic}\ \mathsf{byte}\ \mathsf{byte};$
$\mathsf{putstaticInst} \equiv \mathbf{sputstatic}\ \mathsf{byte}\ \mathsf{byte} \mid \mathbf{bputstatic}\ \mathsf{byte}\ \mathsf{byte};$

– Miscellaneous instructions.
$\mathsf{breakpointInst} \equiv \mathbf{breakpoint};$

We have now completed the definition of the JSP machine structures. To assist the reader retrieving a particular definition, Figures 1 and 2 summarise the read only structures, respectively, the structures that are written to during execution of a JSP program. For each of the three different kinds of structures that we have used, the name is given (followed by an $\equiv$ symbol) and a suggestive graphical representation. The partial maps are shown in a single box, with the domain to the left of the $\not\rightarrow$ symbol and the range to the right. A product data type is shown as a sequence of vertically stacked boxes, one for each component. A sum data type is shown as a horizontally arranged sequence of boxes.

The following sections present the semantic rules for a representative selection of the JSP byte codes. Since there are many groups of similar byte codes, we consider it justified to give the rule for just one member of each group without sacrificing the rigour of the specification.

4.1 Pushing constants onto the stack

The stack is controlled by the stack pointer, which points at the last used location. A short occupies two consecutive locations in the stack, with the high byte at the lowest stack pointer index (bigendian).

The relation $\stackrel{const}{\Rightarrow}$ below describes the effects of each of the instructions dealing with constants on the stack. The type of the relation shows that in addition to the instruction itself, only the stack pointer and the operand stack are relevant here. The left operand of the relation specifies the machine components that are accessed, the right operand mentions those that may be changed by the instruction. Specifying the types of the relations thus provides an aid in the documentation of the system. The types of all relations of the JSP transition system are summarised in Figure 4. We will not give the explicit types of the remaining relations.

$\mathsf{lhs}_{\mathsf{const}} \equiv \langle \mathsf{constInst},\ \mathsf{stackPointer},\ \mathsf{operandStack} \rangle;$
$\mathsf{rhs}_{\mathsf{const}} \equiv \langle \mathsf{stackPointer},\ \mathsf{operandStack} \rangle;$
$\stackrel{const}{\Rightarrow}\ :: (\mathsf{lhs}_{\mathsf{const}} \leftrightarrow \mathsf{rhs}_{\mathsf{const}});$

The rules for **nop**, **bpush** and **spush** below reveal most aspects of the notation that we are using. The semantics of an instruction is defined by an axiom or a rule of inference. The text in square brackets to the left of the axiom/rule is a label to identify the rule. A rule has a number of premises (above the

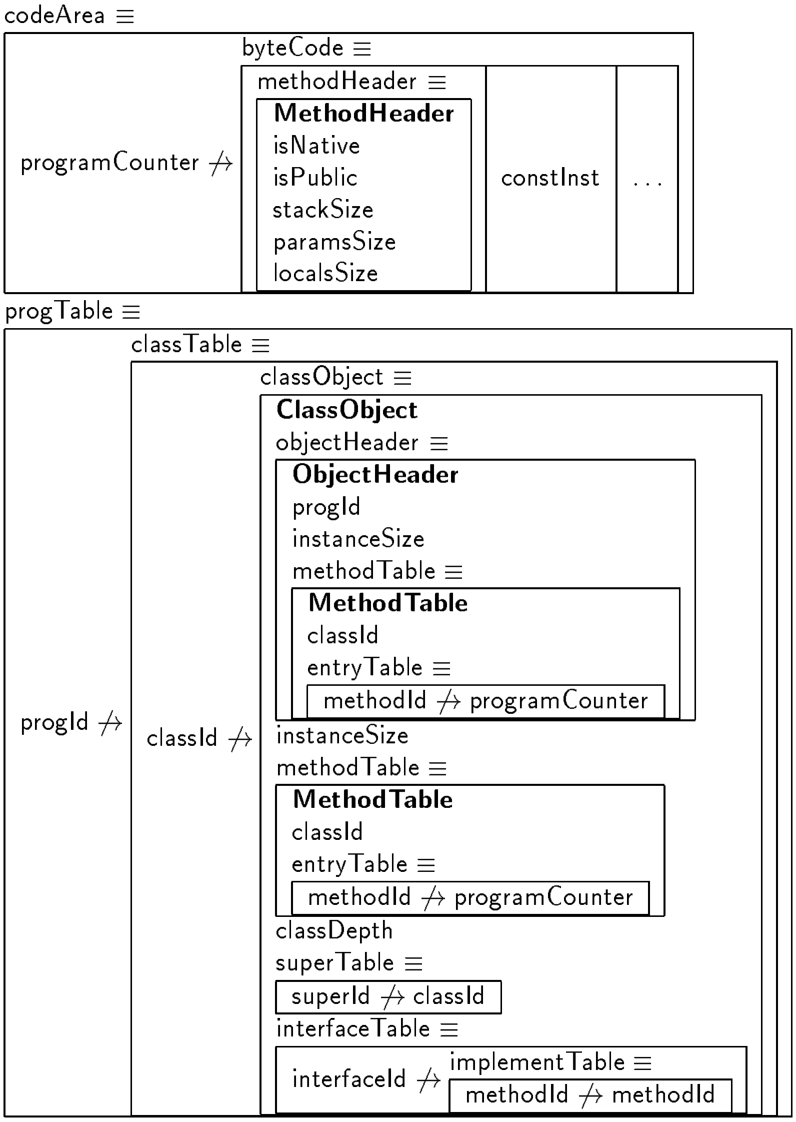

Figure 1. Read only structures

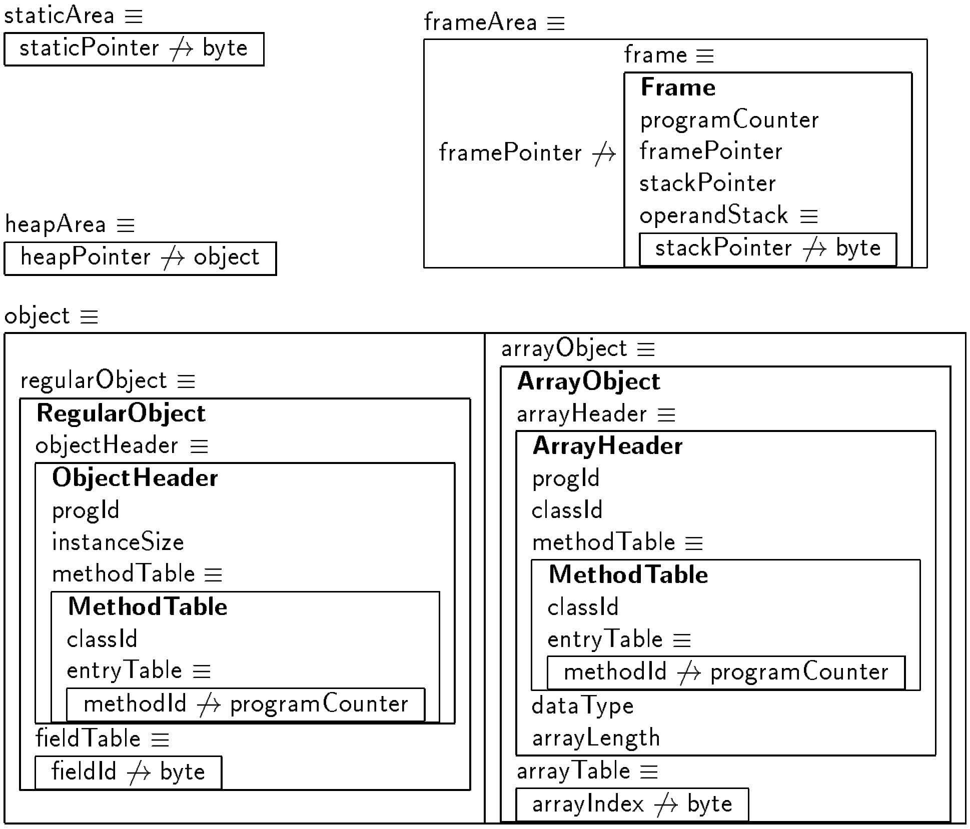

Figure 2. Writeable structures

$\stackrel{ah}{\Rightarrow}$	arrayHeader
$\stackrel{at}{\Rightarrow}$	arrayTable
$\stackrel{b}{\Rightarrow}$	byte
$\stackrel{ct}{\Rightarrow}$	classTable
$\stackrel{fa}{\Rightarrow}$	frameArea
$\stackrel{ft}{\Rightarrow}$	fieldTable

$\stackrel{ha}{\Rightarrow}$	heapArea
$\stackrel{it}{\Rightarrow}$	implementTable
$\stackrel{ob}{\Rightarrow}$	object
$\stackrel{oh}{\Rightarrow}$	objectHeader
$\stackrel{os}{\Rightarrow}$	operandStack
$\stackrel{p}{\Rightarrow}$	(byte, byte)
$\stackrel{ps}{\Rightarrow}$	[(byte, byte)]

$\stackrel{s}{\Rightarrow}$	short
$\stackrel{hp}{\Rightarrow}$	heapPointer
$\stackrel{pc}{\Rightarrow}$	programCounter
$\stackrel{sa}{\Rightarrow}$	staticArea
$\stackrel{f}{\Rightarrow}$	frame
$\stackrel{bc}{\Rightarrow}$	byteCode

Figure 3. Labelled equality relations. The type given is that of the two operands.

horizontal line) and a conclusion. An axiom has a conclusion but no premises. Rules and axioms may have side conditions. The two axioms and the rule below together define the relation $\overset{const}{\Rightarrow}$ over components of the JSP virtual machine configurations.

$$[\mathsf{nop}] \quad \vdash \langle \textbf{nop},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{const}{\Rightarrow} \langle \mathsf{sp},\ \mathsf{os}\rangle;$$

$$[\mathsf{bpush}] \quad \vdash \langle \textbf{bpush}\ \mathsf{v},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{const}{\Rightarrow} \langle \mathsf{sp}+1,\ \mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{v}\}\rangle,$$
$$\textbf{if}\ (\mathsf{sp}+1) \in\ \mathsf{stackPointer}_{\mathsf{range}};$$

$$[\mathsf{spush}] \quad \frac{\vdash \mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp}+2 \mapsto \mathsf{lo}\} \overset{os}{\Rightarrow} \mathsf{os}'}{\vdash \langle \textbf{spush}\ \mathsf{hi}\ \mathsf{lo},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{const}{\Rightarrow} \langle \mathsf{sp}+2,\ \mathsf{os}'\rangle,}$$
$$\textbf{if}\ (\mathsf{sp}+1\ \ldots\ \mathsf{sp}+2) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}};$$

The configuration on the left hand side of the arrow consists of an instruction and its operands (eg. **spush** hi lo), the current stack pointer (sp), and the operand stack (os). Other components of the JSP machine, such as the heap are not used by the three rules above.

The configuration on the right hand side consists of the next value of the stack pointer (eg. $\mathsf{sp}+2$) and the new operand stack (os'). Some of the components mentioned on the left hand side are not present on the right hand side, because they are not changed by the instruction. We have been careful in exposing only the information required, so as to improve the clarity and succinctness of the specification.

The premise of the **spush** rule asserts a relationship between components of the old and the new configuration. The relation $\overset{os}{\Rightarrow}$ is an equality relation, which holds when the operands are both of type operandStack. Labelling equalities with the type of the operands helps the mechanical type checker spot clerical errors. Many other labelled equalities are used throughout. The labels and the types of the operands are summarised in Figure 3. The actual definition of the relations is omitted.

The notation $\mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{v}\}$ extends the mapping os with a new domain/range pair. Any previous association for the new domain value $\mathsf{sp}+1$ is lost. It follows that it is sufficient to decrement the stack pointer to 'forget' mappings for particular values in the domain. Furthermore, we do not in general have the invariant $\mathsf{domain}(\mathsf{os}) = 0 \ldots \mathsf{sp}$.

The side condition for the **bpush** and **spush** operations determines when it is safe to extend the stack. If it is not safe, then the relation $\overset{const}{\Rightarrow}$ does not hold.

The rule for the **apush** operation is not shown here because it is identical to that of the **spush** operation: an address is a numeric value and therefore indistinguishable from a short. In a typed version of the JSP the instructions would not be the same.

4.2 Pushing immediate constants

Some constants are needed so often that special instructions have been defined to push them onto the stack. The semantics of the specialised instructions such as

		programCounter	codeArea	byteCode	stackPointer	operandStack	framePointer	frameArea	heapPointer	heapArea	progId	progTable	staticArea	outputStream
$\stackrel{const}{\Rightarrow}$	constInst				rw	rw								
$\stackrel{load}{\Rightarrow}$	loadInst				rw	rw								
$\stackrel{store}{\Rightarrow}$	storeInst				rw	rw								
$\stackrel{inc}{\Rightarrow}$	incInst				r	rw								
$\stackrel{stack}{\Rightarrow}$	stackInst				rw	rw								
$\stackrel{newarray}{\Rightarrow}$	newarrayInst				r	rw			rw	rw	r	r		
$\stackrel{arrayload}{\Rightarrow}$	arrayloadInst				rw	r				r				
$\stackrel{arraystore}{\Rightarrow}$	arraystoreInst				rw	r				rw				
$\stackrel{arith}{\Rightarrow}$	arithInst				rw	rw								
$\stackrel{logical}{\Rightarrow}$	logicalInst				rw	rw								
$\stackrel{conv}{\Rightarrow}$	convInst				rw	rw								
$\stackrel{compare}{\Rightarrow}$	compareInst				rw	rw								
$\stackrel{control}{\Rightarrow}$	controlInst	rw			rw	rw								
$\stackrel{switch}{\Rightarrow}$	switchInst	rw			rw	rw								
$\stackrel{exception}{\Rightarrow}$	exceptionInst	rw			rw	rw								
$\stackrel{invokeinterface}{\Rightarrow}$	invokeinterfaceInst	rw	r		rw	rw	rw	rw		r	r	r		
$\stackrel{invokevirtual}{\Rightarrow}$	invokevirtualInst	rw	r		rw	rw	rw	rw		r				
$\stackrel{invoke}{\Rightarrow}$	invokeInst	rw	r		rw	rw	rw	rw						
$\stackrel{return}{\Rightarrow}$	returnInst	w			rw	rw	rw	r						
$\stackrel{object}{\Rightarrow}$	objectInst				rw	rw			rw	rw	r	r		
$\stackrel{instance}{\Rightarrow}$	instanceInst				rw	rw			r	r	r	r		
$\stackrel{getfield}{\Rightarrow}$	getfieldInst				rw	r				r				
$\stackrel{putfield}{\Rightarrow}$	putfieldInst				rw	r				rw				
$\stackrel{getstatic}{\Rightarrow}$	getstaticInst				rw	r							r	
$\stackrel{putstatic}{\Rightarrow}$	putstaticInst				rw	r							rw	
$\stackrel{breakpoint}{\Rightarrow}$	breakpointInst				rw	r								rw
$\stackrel{exec}{\Rightarrow}$	execInst	rw	r	r	rw	rw	rw	rw	rw	rw	r	r	rw	rw

Figure 4. A summary of the types of all relations defining the transition system of the Java secure processor.

$\mathbf{bconst_0}$ (below) is defined in terms of the general operation **bpush**. The rules for the remaining instructions $\mathbf{aconst_{null}}$, $\mathbf{bconst_{m1}}$, $\mathbf{bconst_1} \ldots \mathbf{bconst_5}$ (not shown) are defined in a similar way.

$$[\mathsf{bconst_0}]\ \frac{\vdash \langle \mathbf{bpush}\ 0,\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{const}{\Rightarrow} \langle \mathsf{sp}',\ \mathsf{os}'\rangle}{\vdash \langle \mathbf{bconst_0},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{const}{\Rightarrow} \langle \mathsf{sp}',\ \mathsf{os}'\rangle;}$$

4.3 Loading local variables onto the stack

The load instructions transfer values from the parameter and locals area of the stack frame to the top of the operand stack. Local variables and parameters are accessed via a fixed index from the bottom of the operand stack. The reader is reminded that the operand stack is just a portion of the current frame, but we view the operand stack separately from the frame for convenience.

The side conditions on the rules below check for stack overflow. There is no explicit check on the value of the index i because it is assumed that the static semantics of the byte codes, as enforced by the byte code verifier, will deal with illegal offsets.

$$[\mathsf{bload}]\ \frac{\vdash \mathsf{os}(\mathsf{i}) \stackrel{b}{\Rightarrow} \mathsf{v}}{\vdash \langle \mathbf{bload}\ \mathsf{i},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{load}{\Rightarrow} \langle \mathsf{sp}+1,\ \mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{v}\}\rangle,}$$
$$\textbf{if}\ (\mathsf{sp}+1) \in \mathsf{stackPointer_{range}};$$

$$[\mathsf{sload}]\ \frac{\vdash (\mathsf{os}(\mathsf{i}),\ \mathsf{os}(\mathsf{i}+1)) \stackrel{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\quad \vdash \mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp}+2 \mapsto \mathsf{lo}\} \stackrel{os}{\Rightarrow} \mathsf{os}'}{\vdash \langle \mathbf{sload}\ \mathsf{i},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{load}{\Rightarrow} \langle \mathsf{sp}+2,\ \mathsf{os}'\rangle,}$$
$$\textbf{if}\ (\mathsf{sp}+1 \ldots \mathsf{sp}+2) \subseteq \mathsf{stackPointer_{range}};$$

The rule for **aload** and those for the specialised versions $\mathbf{bload_0} \ldots \mathbf{bload_3}$, $\mathbf{sload_0} \ldots \mathbf{sload_3}$ and $\mathbf{aload_0} \ldots \mathbf{aload_3}$ are not shown here.

4.4 Storing stack values into local variables

The store instructions transfer values from the operand stack into parameter and locals area of the stack frame. This time the side conditions check for stack underflow.

$$[\mathsf{bstore}]\ \frac{\vdash \mathsf{os}(\mathsf{sp}) \stackrel{b}{\Rightarrow} \mathsf{v}}{\vdash \langle \mathbf{bstore}\ \mathsf{i},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{store}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{os} \oplus \{\mathsf{i} \mapsto \mathsf{v}\}\rangle,}$$
$$\textbf{if}\ \mathsf{sp} \in \mathsf{stackPointer_{range}};$$

$$[\mathsf{sstore}]\ \frac{\vdash (\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\quad \vdash \mathsf{os} \oplus \{\mathsf{i} \mapsto \mathsf{hi}\} \oplus \{\mathsf{i}+1 \mapsto \mathsf{lo}\} \stackrel{os}{\Rightarrow} \mathsf{os}'}{\vdash \langle \mathbf{sstore}\ \mathsf{i},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{store}{\Rightarrow} \langle \mathsf{sp}-2,\ \mathsf{os}'\rangle,}$$
$$\textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer_{range}};$$

The **astore** instruction is identical to the **sstore** instruction. The specialised instructions $\mathbf{bstore}_0 \ldots \mathbf{bstore}_3$, $\mathbf{sstore}_0 \ldots \mathbf{sstore}_3$ and $\mathbf{astore}_0 \ldots \mathbf{astore}_3$ are not shown here.

$$
\begin{array}{ll}
\mathsf{n2s} & :: \mathsf{num} \rightarrow \mathsf{short}; \\
\mathsf{n2s(n)} & = \mathsf{n}\ \mathit{mod}\ 32768; \\
\mathsf{s2p} & :: \mathsf{short} \rightarrow (\mathsf{byte},\ \mathsf{byte}); \\
\mathsf{s2p(s)} & = (\mathsf{s}\ \mathit{div}\ 256,\ \mathsf{s}\ \mathit{mod}\ 256); \\
\mathsf{b2b} & :: \mathsf{bool} \rightarrow \mathsf{byte}; \\
\mathsf{b2b\ True} & = 1; \\
\mathsf{b2b\ False} & = 0;
\end{array}
\qquad
\begin{array}{ll}
\mathsf{n2b} & :: \mathsf{num} \rightarrow \mathsf{byte}; \\
\mathsf{n2b(n)} & = \mathsf{n}\ \mathit{mod}\ 128; \\
\mathsf{p2s} & :: (\mathsf{byte},\ \mathsf{byte}) \rightarrow \mathsf{short}; \\
\mathsf{p2s(hi,\ lo)} & = 256 * \mathsf{hi} + \mathsf{lo}; \\
== & :: \mathsf{num} \rightarrow \mathsf{num} \rightarrow \mathsf{num}; \\
\mathsf{x}\ ==\ \mathsf{y} & = 1,\ \ \mathbf{if}\ \mathsf{x} > \mathsf{y}; \\
 & = 0,\ \ \mathbf{if}\ \mathsf{x} = \mathsf{y}; \\
 & = -1,\ \ \mathbf{otherwise};
\end{array}
$$

Figure 5. Explicit conversions between arbitrary integers and shorts (n2s), arbitrary integers and bytes (n2b), between shorts and pairs of bytes (s2p, p2s), between booleans and bytes (b2b) and a range comparison operator ==.

4.5 Increment instructions

The increment instructions load the value of a local, increment the value with a signed, 8-bit constant, and store the result. There is no scope for stack underflow or stack overflow, but it is possible for the data to under or overflow. This particular error condition is ignored by the JSP. The specification models this behaviour by using a conversion function n2s, which maps out of bounds values into the range of a short. The functions of Figure 5 define explicit conversions between arbitrary integers and shorts (n2s), arbitrary integers and bytes (n2b), between shorts and pairs of bytes (s2p, p2s), and between booleans and bytes (b2b). These conversions are used consistently throughout the document, so that is would be easy to change the byte order of shorts. This approach makes it easier to implement the JSP on platforms with different views on number representations.

$$
[\mathsf{binc}]\ \frac{\vdash \mathsf{n2b(os(i) + c)} \stackrel{b}{\Rightarrow} \mathsf{v}}{\vdash \langle \mathbf{binc}\ \mathsf{i\ c},\ \mathsf{sp},\ \mathsf{os} \rangle \stackrel{inc}{\Rightarrow} \langle \mathsf{os} \oplus \{\mathsf{i} \mapsto \mathsf{v}\} \rangle;}
$$

$$
[\mathsf{sinc}]\ \frac{\begin{array}{l}\vdash \mathsf{s2p(n2s(p2s(os(i),\ os(i+1)) + c))} \stackrel{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}), \\ \vdash \mathsf{os} \oplus \{\mathsf{i} \mapsto \mathsf{hi}\} \oplus \{\mathsf{i} + 1 \mapsto \mathsf{lo}\} \stackrel{os}{\Rightarrow} \mathsf{os}'\end{array}}{\vdash \langle \mathbf{sinc}\ \mathsf{i\ c},\ \mathsf{sp},\ \mathsf{os} \rangle \stackrel{inc}{\Rightarrow} \langle \mathsf{os}' \rangle;}
$$

4.6 Stack instructions

The stack manipulation instructions are intended to rearrange information on the operand stack. The side conditions check for stack underflow and/or overflow.

- The **pop** and **pop2** instructions remove one, respectively two, bytes from the stack. There are no separate pop instructions for shorts and references, to save opcodes.

[pop^1] $\vdash\langle \textbf{pop},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{stack}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{os}\rangle;$

[pop^2] $\vdash\langle \textbf{pop2},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{stack}{\Rightarrow} \langle \mathsf{sp}-2,\ \mathsf{os}\rangle;$

- The **dup** and **dup2** instructions duplicate one, respectively, two bytes on top of the stack.

$$[\mathsf{dup}]\ \frac{\vdash \mathsf{os}(\mathsf{sp}) \stackrel{b}{\Rightarrow} \mathsf{v}}{\vdash\langle \textbf{dup},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{stack}{\Rightarrow} \langle \mathsf{sp}+1,\ \mathsf{os}\oplus\{\mathsf{sp}+1 \mapsto \mathsf{v}\}\rangle,}$$
$$\textbf{if}\ (\mathsf{sp}\ \ldots\ \mathsf{sp}+1) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}};$$

$$[\mathsf{dup2}]\ \frac{\vdash(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{p}{\Rightarrow} (\mathsf{v}_2,\ \mathsf{v}_1),\quad \vdash \mathsf{os}\oplus\{\mathsf{sp}+1\mapsto \mathsf{v}_2\}\oplus\{\mathsf{sp}+2\mapsto\mathsf{v}_1\} \stackrel{os}{\Rightarrow} \mathsf{os}'}{\vdash\langle \textbf{dup2},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{stack}{\Rightarrow} \langle \mathsf{sp}+2,\ \mathsf{os}'\rangle,}$$
$$\textbf{if}\ (\mathsf{sp}-1\ \ldots\ \mathsf{sp}+2) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}};$$

- The **dup_x** instruction duplicates the top k elements of the operand stack n elements down the stack. The symbol $\uplus$ is the function overriding operator and the notation $\{x_i \mid i \leftarrow [a..b]\}$ generates a set of x_i where i ranges from a to b.

$$[\mathsf{dupx}]\ \frac{\begin{array}{l}\vdash \mathsf{kn}\ mod\ 16 \stackrel{b}{\Rightarrow} \mathsf{n},\\ \vdash \mathsf{kn}\ div\ 16 \stackrel{b}{\Rightarrow} \mathsf{k},\\ \vdash \mathsf{sp}'+\mathsf{k} \stackrel{s}{\Rightarrow} \mathsf{sp}',\\ \vdash \mathsf{os}\ \uplus\{\mathsf{sp}'-\mathsf{i}+1\mapsto \mathsf{os}(\mathsf{sp}-\mathsf{i}+1)\mid \mathsf{i}\leftarrow[\mathsf{n}..1]\} \stackrel{os}{\Rightarrow} \mathsf{os}',\\ \vdash \mathsf{os}'\ \uplus\{\mathsf{sp}'-\mathsf{n}-\mathsf{i}+1\mapsto \mathsf{os}(\mathsf{sp}'-\mathsf{i}+1)\mid \mathsf{i}\leftarrow[\mathsf{k}..1]\} \stackrel{os}{\Rightarrow} \mathsf{os}''\end{array}}{\vdash\langle \textbf{dup\_x}\ \mathsf{kn},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{stack}{\Rightarrow} \langle \mathsf{sp}+\mathsf{k},\ \mathsf{os}''\rangle,}$$
$$\textbf{if}\ (\mathsf{sp}-\mathsf{n}\ \ldots\ \mathsf{sp}+\mathsf{k}) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}}\wedge$$
$$\mathsf{k}\ \in(1\ \ldots\ 4)\wedge\mathsf{n}\ \in(0\ \ldots\ 8)\wedge\mathsf{k}<\mathsf{n};$$

- The **swap** and **swap2** instructions swap the top two bytes, respectively, the top two pairs of bytes on top of the operand stack.

$$[\mathsf{swap}]\ \frac{\vdash(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{p}{\Rightarrow} (\mathsf{v}_2,\ \mathsf{v}_1),\quad \vdash\mathsf{os}\oplus\{\mathsf{sp}-1\mapsto\mathsf{v}_1\}\oplus\{\mathsf{sp}\mapsto\mathsf{v}_2\} \stackrel{os}{\Rightarrow} \mathsf{os}'}{\vdash\langle \textbf{swap},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{stack}{\Rightarrow} \langle \mathsf{sp},\ \mathsf{os}'\rangle,}$$
$$\textbf{if}\ (\mathsf{sp}-1\ \ldots\ \mathsf{sp}) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}};$$

$$[\mathsf{swap2}]\ \frac{\begin{array}{l}\vdash(\mathsf{os}(\mathsf{sp}-3),\ \mathsf{os}(\mathsf{sp}-2)) \stackrel{p}{\Rightarrow} (\mathsf{hi}_2,\ \mathsf{lo}_2),\\ \vdash(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{p}{\Rightarrow} (\mathsf{hi}_1,\ \mathsf{lo}_1),\\ \vdash\mathsf{os}\oplus\{\mathsf{sp}-3\mapsto\mathsf{hi}_1\}\oplus\{\mathsf{sp}-2\mapsto\mathsf{lo}_1\} \stackrel{os}{\Rightarrow} \mathsf{os}',\\ \vdash\mathsf{os}'\oplus\{\mathsf{sp}-1\mapsto\mathsf{hi}_2\}\oplus\{\mathsf{sp}\mapsto\mathsf{lo}_2\} \stackrel{os}{\Rightarrow} \mathsf{os}''\end{array}}{\vdash\langle \textbf{swap2},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{stack}{\Rightarrow} \langle \mathsf{sp},\ \mathsf{os}''\rangle,}$$
$$\textbf{if}\ (\mathsf{sp}-3\ \ldots\ \mathsf{sp}) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}};$$

4.7 Creating array objects

Arrays are stored in the heap. Therefore, the transition relation $\overset{newarray}{\Rightarrow}$ specifies read/write access to the heap, as well as the operand stack. In addition, object creating instructions need to know which is the current application program id (pi). This information is used to classify objects according to who created them. The type of the relation reflects the fact that the program id is used but not changed. (The reader is reminded that Figure 4 summarises the types of all transition relations.)

The array operation **newarray** expects the length of the array on the top of the operand stack. It accesses the length as al. **newarray** creates an appropriate array header ah and a mapping with a domain of $0 \ldots \mathsf{al} - 1$ to serve as the initial value of the array. The method table used is that of class $\mathsf{java.lang.Object}$. The heap is extended with a new object which is to receive the created array header and contents. The reference to the new object is pushed onto the stack. The side condition ensures that stack underflow, heap overflow, or an invalid array length is detected.

$$
[\mathsf{newarray}^1]\;
\frac{
\begin{array}{l}
\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{s}{\Rightarrow} \mathsf{al},\\
\vdash \textbf{ArrayHeader}\ \mathsf{pi}\ 0\ \mathsf{java.lang.Object}_{\mathsf{mt}}\ \textbf{byte}\ \mathsf{al} \overset{ah}{\Rightarrow} \mathsf{ah},\\
\vdash \{\mathsf{i} \mapsto 0 \mid \mathsf{i} \leftarrow [0..\mathsf{al}-1]\} \overset{at}{\Rightarrow} \mathsf{at},\\
\vdash \mathsf{hp} + 1 \overset{hp}{\Rightarrow} \mathsf{hp}',\\
\vdash \mathsf{s2p}(\mathsf{hp}') \overset{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\\
\vdash \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp} \mapsto \mathsf{lo}\} \overset{os}{\Rightarrow} \mathsf{os}',\\
\vdash \mathsf{ha} \oplus \{\mathsf{hp}' \mapsto \textbf{ArrayObject}\ \mathsf{ah}\ \mathsf{at}\} \overset{ha}{\Rightarrow} \mathsf{ha}'
\end{array}
}{
\begin{array}{l}
\vdash \langle \textbf{newarray byte},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt}\rangle\\
\overset{newarray}{\Rightarrow} \langle \mathsf{os}',\ \mathsf{hp}',\ \mathsf{ha}'\rangle,\\
\textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge\\
\ \mathsf{al} \in \mathsf{arrayLength}_{\mathsf{range}} \wedge \mathsf{hp}' \in \mathsf{heapPointer}_{\mathsf{range}};
\end{array}
}
$$

The two other version of **newarray** are not shown here: the **bit** version of **newarray** is identical to the **byte** version above, because each bit is stored in a byte field. The **short** version uses two bytes for storing each short.

The **anewarray** instruction allocates an array of references to objects of the class associated with the given class id (ci). The application program id (pi) is used to access the class table of the current application program. This class table provides the method table for the array elements. The array is initialised to null references.

$$
\begin{array}{ll}
 & \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{s}{\Rightarrow} \mathsf{al}, \\
 & \vdash \mathsf{pt}(\mathsf{pi}) \stackrel{ct}{\Rightarrow} \mathsf{ct}, \\
 & \vdash \mathsf{ct}(\mathsf{ci}) \stackrel{ob}{\Rightarrow} \textbf{ClassObject}\ \_\ \_\ \mathsf{mt}\ \_\ \_\ \_, \\
 & \vdash \textbf{ArrayHeader}\ \mathsf{pi}\ \mathsf{ci}\ \mathsf{mt}\ \textbf{ref}\ \mathsf{al} \stackrel{ah}{\Rightarrow} \mathsf{ah}, \\
 & \vdash \{\mathsf{i} \mapsto 0 \mid \mathsf{i} \leftarrow [0..2*\mathsf{al}-1]\} \stackrel{at}{\Rightarrow} \mathsf{at}, \\
 & \vdash \mathsf{hp}+1 \stackrel{hp}{\Rightarrow} \mathsf{hp}', \\
 & \vdash \mathsf{s2p}(\mathsf{hp}') \stackrel{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}), \\
 & \vdash \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp} \mapsto \mathsf{lo}\} \stackrel{os}{\Rightarrow} \mathsf{os}', \\
 & \vdash \mathsf{ha} \oplus \{\mathsf{hp}' \mapsto \textbf{ArrayObject}\ \mathsf{ah}\ \mathsf{at}\} \stackrel{ha}{\Rightarrow} \mathsf{ha}' \\
\hline
[\mathsf{anewarray}] & \vdash \langle \textbf{anewarray}\ \mathsf{ci},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt}\rangle \\
 & \stackrel{newarray}{\Rightarrow} \langle \mathsf{os}',\ \mathsf{hp}',\ \mathsf{ha}'\rangle, \\
 & \textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge \\
 & \ \mathsf{al} \in \mathsf{arrayLength}_{\mathsf{range}} \wedge \mathsf{hp}' \in \mathsf{heapPointer}_{\mathsf{range}};
\end{array}
$$

4.8 Loading values from arrays

The operation **arraylength** expects an array reference r on the stack and returns the length of the array. The side condition checks for stack underflow, and that a valid heap pointer to an array object is presented.

$$
\begin{array}{ll}
 & \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{s}{\Rightarrow} \mathsf{r}, \\
 & \vdash \mathsf{ha}(\mathsf{r}) \stackrel{ob}{\Rightarrow} \textbf{ArrayObject}(\textbf{ArrayHeader}\ \_\ \_\ \_\ \_\ \mathsf{al})\_, \\
 & \vdash \mathsf{s2p}(\mathsf{al}) \stackrel{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}), \\
 & \vdash \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp} \mapsto \mathsf{lo}\} \stackrel{os}{\Rightarrow} \mathsf{os}' \\
\hline
[\mathsf{arraylength}] & \vdash \langle \textbf{arraylength},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{ha}\rangle \stackrel{arrayload}{\Rightarrow} \langle \mathsf{sp},\ \mathsf{os}'\rangle, \\
 & \textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge \\
 & \ \mathsf{r} \in \mathsf{heapPointer}_{\mathsf{range}} \wedge \mathsf{isArrayObject}(\mathsf{ha}(\mathsf{r}));
\end{array}
$$

Array load instructions access an array and deliver a value at the given index position. The side conditions check for stack underflow, a null reference, an improper object and illegal values of the array index.

$$
\begin{array}{ll}
 & \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-3),\ \mathsf{os}(\mathsf{sp}-2)) \stackrel{s}{\Rightarrow} \mathsf{r}, \\
 & \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{s}{\Rightarrow} \mathsf{i}, \\
 & \vdash \mathsf{ha}(\mathsf{r}) \stackrel{ob}{\Rightarrow} \textbf{ArrayObject}\ \_\ \mathsf{at}, \\
 & \vdash \mathsf{at}(\mathsf{i}) \stackrel{b}{\Rightarrow} \mathsf{v} \\
\hline
[\mathsf{baload}] & \vdash \langle \textbf{baload},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{ha}\rangle \stackrel{arrayload}{\Rightarrow} \langle \mathsf{sp}-3,\ \mathsf{os} \oplus \{\mathsf{sp}-3 \mapsto \mathsf{v}\}\rangle, \\
 & \textbf{if}\ (\mathsf{sp}-3 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge \\
 & \ \mathsf{r} \in \mathsf{heapPointer}_{\mathsf{range}} \wedge \mathsf{isArrayObject}(\mathsf{ha}(\mathsf{r})) \wedge \mathsf{i} \in \mathsf{domain}(\mathsf{at});
\end{array}
$$

$$
\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-3),\ \mathsf{os}(\mathsf{sp}-2)) \overset{s}{\Rightarrow} \mathsf{r},\\ \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{s}{\Rightarrow} \mathsf{i},\\ \vdash \mathsf{ha}(\mathsf{r}) \overset{ob}{\Rightarrow} \textbf{ArrayObject}\ \_\ \mathsf{at},\\ \vdash (\mathsf{at}(\mathsf{i}{*}2),\ \mathsf{at}(\mathsf{i}{*}2+1)) \overset{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}-3 \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp}-2 \mapsto \mathsf{lo}\} \overset{os}{\Rightarrow} \mathsf{os}'\end{array}}{\vdash \langle \textbf{saload},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{ha}\rangle \overset{arrayload}{\Rightarrow} \langle \mathsf{sp}-2,\ \mathsf{os}'\rangle,}\ [\mathsf{saload}]
$$

$$
\begin{array}{l}\textbf{if}\ (\mathsf{sp}-3 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge\\ \ \mathsf{r} \in \mathsf{heapPointer}_{\mathsf{range}} \wedge \mathsf{isArrayObject}(\mathsf{ha}(\mathsf{r})) \wedge\\ (\mathsf{i}{*}2 \ldots \mathsf{i}{*}2+1) \subseteq \mathsf{domain}(\mathsf{at});\end{array}
$$

The operation **aaload** is identical to **saload** and not shown here.

4.9 Storing values into arrays

The array store instructions need read/write access to the stack and read access to the heap. The side conditions check for stack underflow, null references, non-array objects, and illegal array indices. The **aastore** instruction is identical to **sastore**.

$$
\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-4),\ \mathsf{os}(\mathsf{sp}-3)) \overset{s}{\Rightarrow} \mathsf{r},\\ \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-2),\ \mathsf{os}(\mathsf{sp}-1)) \overset{s}{\Rightarrow} \mathsf{i},\\ \vdash \mathsf{os}(\mathsf{sp}) \overset{b}{\Rightarrow} \mathsf{v},\\ \vdash \mathsf{ha}(\mathsf{r}) \overset{ob}{\Rightarrow} \textbf{ArrayObject}\ \mathsf{ah}\ \mathsf{at},\\ \vdash \mathsf{at} \oplus \{\mathsf{i} \mapsto \mathsf{v}\} \overset{at}{\Rightarrow} \mathsf{at}',\\ \vdash \mathsf{ha} \oplus \{\mathsf{r} \mapsto \textbf{ArrayObject}\ \mathsf{ah}\ \mathsf{at}'\} \overset{ha}{\Rightarrow} \mathsf{ha}'\end{array}}{\vdash \langle \textbf{bastore},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{ha}\rangle \overset{arraystore}{\Rightarrow} \langle \mathsf{sp}-5,\ \mathsf{ha}'\rangle,}\ [\mathsf{bastore}]
$$

$$
\begin{array}{l}\textbf{if}\ (\mathsf{sp}-4 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge\\ \ \mathsf{r} \in \mathsf{heapPointer}_{\mathsf{range}} \wedge \mathsf{isArrayObject}(\mathsf{ha}(\mathsf{r})) \wedge \mathsf{i} \in \mathsf{domain}(\mathsf{at});\end{array}
$$

$$
\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-5),\ \mathsf{os}(\mathsf{sp}-4)) \overset{s}{\Rightarrow} \mathsf{r},\\ \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-3),\ \mathsf{os}(\mathsf{sp}-2)) \overset{s}{\Rightarrow} \mathsf{i},\\ \vdash (\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\\ \vdash \mathsf{ha}(\mathsf{r}) \overset{ob}{\Rightarrow} \textbf{ArrayObject}\ \mathsf{ah}\ \mathsf{at},\\ \vdash \mathsf{at} \oplus \{\mathsf{i}{*}2 \mapsto \mathsf{hi}\} \oplus \{\mathsf{i}{*}2+1 \mapsto \mathsf{lo}\} \overset{at}{\Rightarrow} \mathsf{at}',\\ \vdash \mathsf{ha} \oplus \{\mathsf{r} \mapsto \textbf{ArrayObject}\ \mathsf{ah}\ \mathsf{at}'\} \overset{ha}{\Rightarrow} \mathsf{ha}'\end{array}}{\vdash \langle \textbf{sastore},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{ha}\rangle \overset{arraystore}{\Rightarrow} \langle \mathsf{sp}-6,\ \mathsf{ha}'\rangle,}\ [\mathsf{sastore}]
$$

$$
\begin{array}{l}\textbf{if}\ (\mathsf{sp}-5 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge\\ \ \mathsf{r} \in \mathsf{heapPointer}_{\mathsf{range}} \wedge \mathsf{isArrayObject}(\mathsf{ha}(\mathsf{r})) \wedge\\ (\mathsf{i}{*}2 \ldots \mathsf{i}{*}2+1) \subseteq \mathsf{domain}(\mathsf{at});\end{array}
$$

4.10 Arithmetic

The unary (arithmetic) negation operator is defined below for bytes and shorts. It ignores under/overflow of values, but checks for stack underflow.

$$[\mathrm{bneg}]\ \frac{\vdash \mathrm{os}(\mathrm{sp}) \overset{b}{\Rightarrow} \mathrm{v}}{\begin{array}{l}\vdash \langle \mathbf{bneg},\ \mathrm{sp},\ \mathrm{os}\rangle \overset{arith}{\Rightarrow} \langle \mathrm{sp},\ \mathrm{os} \oplus \{\mathrm{sp} \mapsto \mathrm{n2b}(-\mathrm{v})\}\rangle, \\ \mathbf{if}\ \mathrm{sp}\ \in\ \mathrm{stackPointer}_{\mathrm{range}};\end{array}}$$

$$[\mathrm{sneg}]\ \frac{\begin{array}{l}\vdash \mathrm{s2p}(\mathrm{n2s}(-(\mathrm{p2s}(\mathrm{os}(\mathrm{sp}-1),\ \mathrm{os}(\mathrm{sp}))))) \overset{p}{\Rightarrow} (\mathrm{hi},\ \mathrm{lo}), \\ \vdash \mathrm{os} \oplus \{\mathrm{sp}-1 \mapsto \mathrm{hi}\} \oplus \{\mathrm{sp} \mapsto \mathrm{lo}\} \overset{os}{\Rightarrow} \mathrm{os}'\end{array}}{\begin{array}{l}\vdash \langle \mathbf{sneg},\ \mathrm{sp},\ \mathrm{os}\rangle \overset{arith}{\Rightarrow} \langle \mathrm{sp},\ \mathrm{os}'\rangle, \\ \mathbf{if}\ (\mathrm{sp}-1 \ldots \mathrm{sp}) \subseteq\ \mathrm{stackPointer}_{\mathrm{range}};\end{array}}$$

Binary addition for bytes and shorts is defined below. The other binary arithmetic instructions (for subtraction, multiplication, division and remainder) are defined in the same way, and are not shown. The side condition of the division and remainder operations check that the divisor is non-zero.

$$[\mathrm{badd}]\ \frac{\vdash (\mathrm{os}(\mathrm{sp}-1),\ \mathrm{os}(\mathrm{sp})) \overset{p}{\Rightarrow} (\mathrm{v}_2,\ \mathrm{v}_1)}{\begin{array}{l}\vdash \langle \mathbf{badd},\ \mathrm{sp},\ \mathrm{os}\rangle \overset{arith}{\Rightarrow} \langle \mathrm{sp}-1,\ \mathrm{os} \oplus \{\mathrm{sp}-1 \mapsto \mathrm{n2b}(\mathrm{v}_2+\mathrm{v}_1)\}\rangle, \\ \mathbf{if}\ (\mathrm{sp}-1 \ldots \mathrm{sp}) \subseteq\ \mathrm{stackPointer}_{\mathrm{range}};\end{array}}$$

$$[\mathrm{sadd}]\ \frac{\begin{array}{l}\vdash \mathrm{p2s}(\mathrm{os}(\mathrm{sp}-3),\ \mathrm{os}(\mathrm{sp}-2)) \overset{s}{\Rightarrow} \mathrm{v}_2, \\ \vdash \mathrm{p2s}(\mathrm{os}(\mathrm{sp}-1),\ \mathrm{os}(\mathrm{sp})) \overset{s}{\Rightarrow} \mathrm{v}_1, \\ \vdash \mathrm{s2p}(\mathrm{n2s}(\mathrm{v}_2+\mathrm{v}_1)) \overset{p}{\Rightarrow} (\mathrm{hi},\ \mathrm{lo}), \\ \vdash \mathrm{os} \oplus \{\mathrm{sp}-3 \mapsto \mathrm{hi}\} \oplus \{\mathrm{sp}-2 \mapsto \mathrm{lo}\} \overset{os}{\Rightarrow} \mathrm{os}'\end{array}}{\begin{array}{l}\vdash \langle \mathbf{sadd},\ \mathrm{sp},\ \mathrm{os}\rangle \overset{arith}{\Rightarrow} \langle \mathrm{sp}-2,\ \mathrm{os}'\rangle, \\ \mathbf{if}\ (\mathrm{sp}-3 \ldots \mathrm{sp}) \subseteq\ \mathrm{stackPointer}_{\mathrm{range}};\end{array}}$$

4.11 Logical instructions

The logical shift left as defined below shifts the element next to the top of the stack. The shift count is the top of the stack. The remaining binary logical instructions (for arithmetic shift right with sign extension, unsigned shift right, bit-wise and, bit-wise or and bit-wise exclusive or) are defined in the same way and are not shown.

$$[\mathrm{bshl}]\ \frac{\vdash (\mathrm{os}(\mathrm{sp}-1),\ \mathrm{os}(\mathrm{sp})) \overset{p}{\Rightarrow} (\mathrm{v}_2,\ \mathrm{v}_1)}{\begin{array}{l}\vdash \langle \mathbf{bshl},\ \mathrm{sp},\ \mathrm{os}\rangle \overset{logical}{\Rightarrow} \langle \mathrm{sp}-1,\ \mathrm{os} \oplus \{\mathrm{sp}-1 \mapsto \mathrm{n2b}(\mathrm{v}_2\ \ll\ \mathrm{v}_1)\}\rangle, \\ \mathbf{if}\ (\mathrm{sp}-1 \ldots \mathrm{sp}) \subseteq\ \mathrm{stackPointer}_{\mathrm{range}} \wedge \mathrm{v}_1\ \in (0 \ldots 7);\end{array}}$$

$$[\mathrm{sshl}]\ \frac{\begin{array}{l}\vdash \mathrm{p2s}(\mathrm{os}(\mathrm{sp}-3),\ \mathrm{os}(\mathrm{sp}-2)) \overset{s}{\Rightarrow} \mathrm{v}_2, \\ \vdash \mathrm{p2s}(\mathrm{os}(\mathrm{sp}-1),\ \mathrm{os}(\mathrm{sp})) \overset{s}{\Rightarrow} \mathrm{v}_1, \\ \vdash \mathrm{s2p}(\mathrm{n2s}(\mathrm{v}_2\ \ll\ \mathrm{v}_1)) \overset{p}{\Rightarrow} (\mathrm{hi},\ \mathrm{lo}), \\ \vdash \mathrm{os} \oplus \{\mathrm{sp}-3 \mapsto \mathrm{hi}\} \oplus \{\mathrm{sp}-2 \mapsto \mathrm{lo}\} \overset{os}{\Rightarrow} \mathrm{os}'\end{array}}{\begin{array}{l}\vdash \langle \mathbf{sshl},\ \mathrm{sp},\ \mathrm{os}\rangle \overset{logical}{\Rightarrow} \langle \mathrm{sp}-2,\ \mathrm{os}'\rangle, \\ \mathbf{if}\ (\mathrm{sp}-3 \ldots \mathrm{sp}) \subseteq\ \mathrm{stackPointer}_{\mathrm{range}} \wedge \mathrm{v}_1\ \in (0 \ldots 15);\end{array}}$$

4.12 Conversions

The conversion operations explicitly truncate a short to a byte or zero fill a byte to a short. Stack underflow and overflow are detected.

$$[\mathsf{s2b}]\ \frac{\vdash \mathsf{n2b}(\mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp}))) \stackrel{s}{\Rightarrow} \mathsf{v}}{\vdash\langle \mathbf{s2b},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{conv}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{os}\oplus\{\mathsf{sp}-1\mapsto \mathsf{v}\}\rangle,}\ \mathbf{if}\ (\mathsf{sp}-1\ \ldots\ \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}};$$

$$[\mathsf{b2s}]\ \frac{\vdash \mathsf{s2p}(\mathsf{os}(\mathsf{sp})) \stackrel{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\quad \vdash \mathsf{os}\oplus\{\mathsf{sp}\mapsto \mathsf{hi}\}\oplus\{\mathsf{sp}+1\mapsto \mathsf{lo}\} \stackrel{os}{\Rightarrow} \mathsf{os}'}{\vdash\langle \mathbf{b2s},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{conv}{\Rightarrow} \langle \mathsf{sp}+1,\ \mathsf{os}'\rangle,}\ \mathbf{if}\ (\mathsf{sp}\ \ldots\ \mathsf{sp}+1) \subseteq \mathsf{stackPointer}_{\mathsf{range}};$$

4.13 Comparisons

The compare instruction **bcmp** returns -1 if the top element of the stack is greater than the one below it. It returns 0 if the top two elements are equal and 1 otherwise. The **scmp** instruction compares the shorts on top of the stack. The definition of the range comparison operator $==$ is given in Figure 5.

$$[\mathsf{bcmp}]\ \frac{\vdash (\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{p}{\Rightarrow} (\mathsf{v}_2,\ \mathsf{v}_1)}{\vdash\langle \mathbf{bcmp},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{compare}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{os}\oplus\{\mathsf{sp}-1\mapsto (\mathsf{v}_2\ ==\ \mathsf{v}_1)\}\rangle,}\ \mathbf{if}\ (\mathsf{sp}-1\ \ldots\ \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}};$$

$$[\mathsf{scmp}]\ \frac{\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-3),\ \mathsf{os}(\mathsf{sp}-2)) \stackrel{s}{\Rightarrow} \mathsf{v}_2,\quad \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{s}{\Rightarrow} \mathsf{v}_1,\quad \vdash \mathsf{os}\oplus\{\mathsf{sp}-3\mapsto (\mathsf{v}_2\ ==\ \mathsf{v}_1)\} \stackrel{os}{\Rightarrow} \mathsf{os}'}{\vdash\langle \mathbf{scmp},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{compare}{\Rightarrow} \langle \mathsf{sp}-3,\ \mathsf{os}'\rangle,}\ \mathbf{if}\ (\mathsf{sp}-3\ \ldots\ \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}};$$

The **acmp** instruction compares object references and returns 0 if the references are equal, 1 otherwise.

$$[\mathsf{acmp}]\ \frac{\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-3),\ \mathsf{os}(\mathsf{sp}-2)) \stackrel{s}{\Rightarrow} \mathsf{v}_2,\quad \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{s}{\Rightarrow} \mathsf{v}_1}{\vdash\langle \mathbf{acmp},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{compare}{\Rightarrow} \langle \mathsf{sp}-3,\ \mathsf{os}\oplus\{\mathsf{sp}-3\mapsto (\mathsf{v}_2\ ==\ \mathsf{v}_1) \mathit{mod}\ 2\}\rangle,}\ \mathbf{if}\ (\mathsf{sp}-3\ \ldots\ \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}};$$

4.14 Transferring control

The **ifeq** instruction adds its immediate operand to the value of the program counter (pc) if the top of the stack contains 0. Otherwise the program counter is incremented to point at the next instruction. Stack underflow is detected. The static semantics is assumed to detect illegal values for the program counter.

$$\frac{\vdash \mathsf{os}(\mathsf{sp}) \stackrel{b}{\Rightarrow} \mathsf{v}, \quad \vdash \mathsf{pc} + \mathsf{p2s\ offset} \stackrel{pc}{\Rightarrow} \mathsf{pc}'}{\vdash \langle \mathsf{pc},\ \textbf{ifeq}\ \mathsf{offset},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{control}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}-1,\ \mathsf{os}\rangle,} \quad [\mathsf{ifeq}^0]$$
$$\textbf{if}\ \mathsf{sp}\ \in\ \mathsf{stackPointer}_{\mathsf{range}} \wedge \mathsf{v} = 0;$$

$$\frac{\vdash \mathsf{os}(\mathsf{sp}) \stackrel{b}{\Rightarrow} \mathsf{v}}{\vdash \langle \mathsf{pc},\ \textbf{ifeq}\ \mathsf{offset},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{control}{\Rightarrow} \langle \mathsf{pc}+1,\ \mathsf{sp}-1,\ \mathsf{os}\rangle,} \quad [\mathsf{ifeq}^1]$$
$$\textbf{if}\ \mathsf{sp}\ \in\ \mathsf{stackPointer}_{\mathsf{range}} \wedge \mathsf{v} \neq 0;$$

The remaining operations **iflt**, **ifgt**, **ifne**, **ifge**, **ifle** are similar and not shown.

The static semantics is assumed to check that the unconditional jump instruction **goto** carries a valid offset.

$$\frac{\vdash \mathsf{pc} + \mathsf{p2s\ offset} \stackrel{s}{\Rightarrow} \mathsf{pc}'}{\vdash \langle \mathsf{pc},\ \textbf{goto}\ \mathsf{offset},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{control}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp},\ \mathsf{os}\rangle;} \quad [\mathsf{goto}]$$

4.15 Support for switch statements

The **tableswitch** and **lookupswitch** instructions provide support for the Java switch satements. The **tableswitch** instruction allows for a selection of jump targets from an indexed table, with the choice index coming from the stack. The **lookupswitch** instruction is similar, except that a keyed table is used rather than an indexd one.

Both instructions have a number of immediate operands, the first of which is the default offset. The **tableswitch** instruction has further immediate operands to specify the lower and upperbounds of a jump table and the jump table itself. The instruction expects a byte index on the stack, which is used to select the appropriate offset from the jump table. The offset is then added to the current value of the program counter. If the index lies outside the range defined by the lower and upperbound, the default offset is added to the program counter.

The side condition checks that the stack pointer is valid, but does not need to check that the old or new values of the program counter are valid. This is the task of the static semantics.

$$\frac{\vdash \mathsf{os}(\mathsf{sp}) \stackrel{b}{\Rightarrow} \mathsf{index}, \quad \vdash \mathsf{cases}(\mathsf{index}) \stackrel{p}{\Rightarrow} \mathsf{offset}, \quad \vdash \mathsf{pc} + \mathsf{p2s}(\mathsf{offset}) \stackrel{s}{\Rightarrow} \mathsf{pc}'}{\vdash \langle \mathsf{pc},\ \textbf{tableswitch}\ \mathsf{default\ low\ high\ cases},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{switch}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}-1,\ \mathsf{os}\rangle,} \quad [\mathsf{tableswitch}^1]$$
$$\textbf{if}\ \mathsf{sp}\ \in\ \mathsf{stackPointer}_{\mathsf{range}} \wedge \mathsf{index}\ \in (\mathsf{low}\ \ldots\ \mathsf{high});$$

$$\text{[tableswitch}^2\text{]}\ \frac{\vdash \mathsf{os(sp)} \overset{b}{\Rightarrow} \mathsf{index},\quad \vdash \mathsf{pc} + \mathsf{p2s(default)} \overset{s}{\Rightarrow} \mathsf{pc}'}{\vdash \langle \mathsf{pc},\ \textbf{tableswitch}\ \mathsf{default\ low\ high\ cases},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{switch}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp} - 1,\ \mathsf{os}\rangle,}$$
$$\textbf{if}\ \mathsf{sp} \in \mathsf{stackPointer}_{\mathsf{range}} \wedge \mathsf{index} \notin (\mathsf{low} \ldots \mathsf{high});$$

The **lookupswitch** has a default offset and further immediate operands to specify the number of entries in the jump table and the jump table itself. The **lookupswitch** instruction expects a key on the stack, which when it occurs in the table is used to select the appropriate offset from the jump table. The offset is then added to the current value of the program counter. If the key does not occur in the jump table, the default offset is added to the program counter.

$$\text{[lookupswitch}^1\text{]}\ \frac{\vdash \mathsf{os(sp)} \overset{b}{\Rightarrow} \mathsf{key},\quad \vdash \{\mathsf{o} \mid (\mathsf{k},\ \mathsf{o}) \leftarrow \mathsf{range(cases)} \wedge \ \mathsf{key} = \mathsf{k}\} \overset{ps}{\Rightarrow} \mathsf{offsets},\quad \vdash \mathsf{pc} + \mathsf{p2s(hd(offsets))} \overset{s}{\Rightarrow} \mathsf{pc}'}{\vdash \langle \mathsf{pc},\ \textbf{lookupswitch}\ \mathsf{default\ entries\ cases},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{switch}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp} - 1,\ \mathsf{os}\rangle,}$$
$$\textbf{if}\ \mathsf{sp} \in \mathsf{stackPointer}_{\mathsf{range}} \wedge \mathsf{offsets} \neq \{\};$$

$$\text{[lookupswitch}^2\text{]}\ \frac{\vdash \mathsf{os(sp)} \overset{b}{\Rightarrow} \mathsf{key},\quad \vdash \{\mathsf{o} \mid (\mathsf{k},\ \mathsf{o}) \leftarrow \mathsf{range(cases)} \wedge \ \mathsf{key} = \mathsf{k}\} \overset{ps}{\Rightarrow} \mathsf{offsets},\quad \vdash \mathsf{pc} + \mathsf{p2s(default)} \overset{s}{\Rightarrow} \mathsf{pc}'}{\vdash \langle \mathsf{pc},\ \textbf{lookupswitch}\ \mathsf{default\ entries\ cases},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{switch}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp} - 1,\ \mathsf{os}\rangle,}$$
$$\textbf{if}\ \mathsf{sp} \in \mathsf{stackPointer}_{\mathsf{range}} \wedge \mathsf{offsets} = \{\};$$

4.16 Exception handling

The **athrow** instruction terminates the execution of the JSP program, for there is no pc, sp and os for which the relation below holds. The present treatment of exceptions is somewhat crude, but consistent with ISO 7816-4 requirements.

$$\text{[athrow]}\ \vdash \langle \mathsf{pc},\ \textbf{athrow},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{exception}{\Rightarrow} \langle \mathsf{pc},\ \mathsf{sp},\ \mathsf{os}\rangle,$$
$$\textbf{if}\ \mathsf{False};$$

The **jsr** and **ret** instructions are used by the JVM to support exception handling. Even though the JSP provides only rudimentary support for exceptions, the semantics of these two instructions is well defined. Stack overflow and illegal return addresses are detected.

$$\text{[ret]}\ \frac{\vdash \mathsf{p2s(os(i),\ os(i+1))} \overset{s}{\Rightarrow} \mathsf{pc}'}{\vdash \langle \mathsf{pc},\ \textbf{ret}\ \mathsf{i},\ \mathsf{sp},\ \mathsf{os}\rangle \overset{exception}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp},\ \mathsf{os}\rangle,}$$
$$\textbf{if}\ \mathsf{pc}' \in \mathsf{programCounter}_{\mathsf{range}};$$

$$\frac{\begin{array}{l}\vdash \mathsf{pc} + \mathsf{p2s}(\mathsf{hi_v},\ \mathsf{lo_v}) \overset{s}{\Rightarrow} \mathsf{pc}',\\ \vdash \mathsf{s2p}(\mathsf{pc}) \overset{p}{\Rightarrow} (\mathsf{hi_p},\ \mathsf{lo_p}),\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{hi_p}\} \oplus \{\mathsf{sp}+2 \mapsto \mathsf{lo_p}\} \overset{os}{\Rightarrow} \mathsf{os}'\end{array}}{\begin{array}{l}\vdash \langle \mathsf{pc},\ \mathbf{jsr}(\mathsf{hi_v},\ \mathsf{lo_v}),\ \mathsf{sp},\ \mathsf{os}\rangle \overset{exception}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}+2,\ \mathsf{os}'\rangle,\\ \textbf{if } (\mathsf{sp}+1 \ldots \mathsf{sp}+2) \subseteq\ \mathsf{stackPointer_{range}} \wedge\\ \ \mathsf{pc}' \ \in\ \mathsf{programCounter_{range}};\end{array}}\ [\mathsf{jsr}]$$

4.17 Method invocation

The JSP has three different instruction to invoke methods. The **invokevirtual** is the normal dynamic method dispatch instruction. The **invoke** instruction is used when the Java compiler or JSP to JVM byte code translator are able to determine statically which method to invoke. The **invokeinterface** instruction supports Java's approach to multiple inheritance by searching for a method that implements an abstract method from an interface.

The **invokeinterface** instruction has three operands. The first, params, specifies the number of arguments to be expected on the operand stack. The second immediate operand, ii, indicates the index of an interface. The third mi determines which (abstract) method within the interface is required.

$$\frac{\begin{array}{l}\vdash \mathsf{pt}(\mathsf{pi}) \overset{ct}{\Rightarrow} \mathsf{ct},\\ \vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-\mathsf{params}+1),\ \mathsf{os}(\mathsf{sp}-\mathsf{params}+2)) \overset{s}{\Rightarrow} \mathsf{r},\\ \vdash \mathsf{ha}(\mathsf{r}) \overset{ob}{\Rightarrow} \textbf{RegularObject}\ \mathsf{oh}\ \_,\\ \vdash \mathsf{oh} \overset{oh}{\Rightarrow} \textbf{ObjectHeader}\ \_\ \_(\textbf{MethodTable}\ \mathsf{ci}\ \mathsf{et}),\\ \vdash \mathsf{ct}(\mathsf{ci}) \overset{ob}{\Rightarrow} \textbf{ClassObject}\ \_\ \_\ \_\ \_\ \_\ \mathsf{cit},\\ \vdash \mathsf{cit}(\mathsf{ii}) \overset{it}{\Rightarrow} \mathsf{it},\\ \vdash \mathsf{it}(\mathsf{mi}) \overset{s}{\Rightarrow} \mathsf{mi}',\\ \vdash \mathsf{et}(\mathsf{mi}') \overset{s}{\Rightarrow} \mathsf{pc}',\\ \vdash \{\mathsf{params}-\mathsf{i} \mapsto \mathsf{os}(\mathsf{sp}-\mathsf{i}+1) \mid \mathsf{i} \leftarrow [1..\mathsf{params}]\} \overset{os}{\Rightarrow} \mathsf{os}',\\ \vdash \mathsf{fa} \oplus \{\mathsf{fp}+1 \mapsto \textbf{Frame}(\mathsf{pc}+1)\mathsf{fp}(\mathsf{sp}-\mathsf{params})\mathsf{os}\} \overset{fa}{\Rightarrow} \mathsf{fa}'\end{array}}{\begin{array}{l}\vdash \langle \mathsf{pc},\ \mathsf{ca},\ \textbf{invokeinterface}\ \mathsf{params}\ \mathsf{ii}\ \mathsf{mi},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt}\rangle\\ \overset{invokeinterface}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{params}-1,\ \mathsf{os}',\ \mathsf{fp}+1,\ \mathsf{fa}'\rangle,\\ \textbf{if } (\mathsf{sp}-\mathsf{params}+1 \ldots \mathsf{sp}) \subseteq\ \mathsf{stackPointer_{range}} \wedge\\ \ \mathsf{r}\ \in\ \mathsf{heapPointer_{range}} \wedge \mathsf{isRegularObject}(\mathsf{ha}(\mathsf{r})) \wedge\\ \ \mathsf{ci}\ \in\ \mathsf{classId_{range}} \wedge \mathsf{ii}\ \in\ \mathsf{interfaceId_{range}} \wedge\\ \ \mathsf{mi}\ \in\ \mathsf{methodId_{range}} \wedge \mathsf{mi}'\ \in\ \mathsf{methodId_{range}} \wedge\\ \ \mathsf{fp}+1\ \in\ \mathsf{framePointer_{range}};\end{array}}\ [\mathsf{invoke}^1]$$

The top of the operand stack must contain a reference to an object, which should be an instance of a regular class that implements the interface method. The header of the object is accessed to yield the interface table (cit) associated with the object. The table it maps the method index of the abstract method (mi) onto the method index of the implementation (mi′). The latter is then used to locate the appropriate program counter in the method table of the object pointed at by r. The value of the program counter will be made to point at the first proper instruction of the method. A new frame is created, linking to the previous frame for the benefit of the return instruction. Execution continues at the first instruction of the callee.

The **invokevirtual** instruction expects a reference to an object on top of the operand stack. The object header of the object is accessed to yield the method table associated with the object. The method index mi determines which method is to be activated. The value of the program counter pc will be made to point at the first proper instruction of the method. A new frame is created, linking to the previous frame for the benefit of the return instruction. Execution continues at the first instruction of the callee.

$$[\mathsf{invoke}^2]\ \frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-\mathsf{params}+1),\ \mathsf{os}(\mathsf{sp}-\mathsf{params}+2)) \stackrel{s}{\Rightarrow} \mathsf{r},\\ \vdash \mathsf{ha}(\mathsf{r}) \stackrel{ob}{\Rightarrow} \textbf{RegularObject}\ \mathsf{oh}\ \_,\\ \vdash \mathsf{oh} \stackrel{oh}{\Rightarrow} \textbf{ObjectHeader}\ \_\ \_(\textbf{MethodTable}\ \_\ \mathsf{et}),\\ \vdash \mathsf{et}(\mathsf{mi}) \stackrel{s}{\Rightarrow} \mathsf{pc}',\\ \vdash \{\mathsf{params}-\mathsf{i} \mapsto \mathsf{os}(\mathsf{sp}-\mathsf{i}+1)\mid \mathsf{i}\leftarrow[1..\mathsf{params}]\} \stackrel{os}{\Rightarrow} \mathsf{os}',\\ \vdash \mathsf{fa} \oplus \{\mathsf{fp}+1 \mapsto \textbf{Frame}(\mathsf{pc}+1)\mathsf{fp}(\mathsf{sp}-\mathsf{params})\mathsf{os}\} \stackrel{fa}{\Rightarrow} \mathsf{fa}'\end{array}}{\begin{array}{l}\vdash\langle \mathsf{pc},\ \mathsf{ca},\ \textbf{invokevirtual}\ \mathsf{params}\ \mathsf{mi},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa},\ \mathsf{ha}\rangle\\ \stackrel{invokevirtual}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{params}-1,\ \mathsf{os}',\ \mathsf{fp}+1,\ \mathsf{fa}'\rangle,\\ \textbf{if}\ (\mathsf{sp}-\mathsf{params}+1\ \ldots\ \mathsf{sp}) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}} \wedge\\ \ \mathsf{r}\ \in\ \mathsf{heapPointer}_{\mathsf{range}} \wedge \mathsf{isRegularObject}(\mathsf{ha}(\mathsf{r}))\wedge\\ \ \mathsf{mi}\ \in\ \mathsf{methodId}_{\mathsf{range}} \wedge \mathsf{fp}+1\ \in\ \mathsf{framePointer}_{\mathsf{range}};\end{array}}$$

The immediate operands of the **invoke** instruction specify the two bytes that determine the index of the method in the $\mathsf{codeArea}$. The number of parameters is retrieved from the method header (which is stored in the pseudo instruction preceding the first proper instruction of the method).

$$[\mathsf{invoke}^3]\ \frac{\begin{array}{l}\vdash \mathsf{p2s}\ \mathsf{offset} \stackrel{s}{\Rightarrow} \mathsf{pc}',\\ \vdash \mathsf{ca}(\mathsf{pc}'-1) \stackrel{bc}{\Rightarrow} (\textbf{MethodHeader}\ \_\ \_\ \_\ \mathsf{params}\ \mathsf{locals}),\\ \vdash \{\mathsf{params}-\mathsf{i} \mapsto \mathsf{os}(\mathsf{sp}+1-\mathsf{i})\mid \mathsf{i}\leftarrow[1..\mathsf{params}]\} \stackrel{os}{\Rightarrow} \mathsf{os}',\\ \vdash \mathsf{fa} \oplus \{\mathsf{fp}+1 \mapsto \textbf{Frame}(\mathsf{pc}+1)\mathsf{fp}(\mathsf{sp}-\mathsf{params})\mathsf{os}\} \stackrel{fa}{\Rightarrow} \mathsf{fa}'\end{array}}{\begin{array}{l}\vdash\langle \mathsf{pc},\ \mathsf{ca},\ \textbf{invoke}\ \mathsf{offset},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa}\rangle\\ \stackrel{invoke}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{locals}+\mathsf{params}-1,\ \mathsf{os}',\ \mathsf{fp}+1,\ \mathsf{fa}'\rangle,\\ \textbf{if}\ (\mathsf{sp}-\mathsf{params}+1\ \ldots\ \mathsf{sp}) \subseteq\ \mathsf{stackPointer}_{\mathsf{range}} \wedge\\ \ \mathsf{fp}+1\ \in\ \mathsf{framePointer}_{\mathsf{range}};\end{array}}$$

4.18 Method return

The return instructions below return from a (non-static) method. The four instructions differ only in the return value produced. Each return instruction abandons the frame pointed at by the frame pointer and returns to the previous frame pointer. The appropriate return value is deposited onto the operand stack of the caller (except in the last case below, which is intended for a void returning method). The side conditions check for stack under/overflow and frame underflow.

$$\frac{\begin{array}{l}\vdash \mathsf{fa}(\mathsf{fp}) \stackrel{f}{\Rightarrow} \textbf{Frame}\ \mathsf{pc}'\ \mathsf{fp}'\ \mathsf{sp}'\ \mathsf{os}',\\ \vdash \mathsf{os}(\mathsf{sp}) \stackrel{b}{\Rightarrow} \mathsf{v},\\ \vdash \mathsf{os}' \oplus \{\mathsf{sp}'+1 \mapsto \mathsf{v}\} \stackrel{os}{\Rightarrow} \mathsf{os}''\end{array}}{\begin{array}{l}\vdash \langle \textbf{breturn},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa}\rangle \stackrel{return}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}'+1,\ \mathsf{os}'',\ \mathsf{fp}'\rangle,\\ \textbf{if}\ \mathsf{fp} \in \mathsf{framePointer}_{\mathsf{range}} \wedge \mathsf{sp} \in \mathsf{stackPointer}_{\mathsf{range}} \wedge\\ (\mathsf{sp}'+1) \in \mathsf{stackPointer}_{\mathsf{range}};\end{array}}\ [\mathsf{breturn}]$$

$$\frac{\begin{array}{l}\vdash \mathsf{fa}(\mathsf{fp}) \stackrel{f}{\Rightarrow} \textbf{Frame}\ \mathsf{pc}'\ \mathsf{fp}'\ \mathsf{sp}'\ \mathsf{os}',\\ \vdash (\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\\ \vdash \mathsf{os}' \oplus \{\mathsf{sp}'+1 \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp}'+2 \mapsto \mathsf{lo}\} \stackrel{os}{\Rightarrow} \mathsf{os}''\end{array}}{\begin{array}{l}\vdash \langle \textbf{sreturn},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa}\rangle \stackrel{return}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}'+2,\ \mathsf{os}'',\ \mathsf{fp}'\rangle,\\ \textbf{if}\ \mathsf{fp} \in \mathsf{framePointer}_{\mathsf{range}} \wedge\\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge\\ (\mathsf{sp}'+1 \ldots \mathsf{sp}'+2) \subseteq \mathsf{stackPointer}_{\mathsf{range}};\end{array}}\ [\mathsf{sreturn}]$$

$$\frac{\vdash \mathsf{fa}(\mathsf{fp}) \stackrel{f}{\Rightarrow} \textbf{Frame}\ \mathsf{pc}'\ \mathsf{fp}'\ \mathsf{sp}'\ \mathsf{os}'}{\begin{array}{l}\vdash \langle \textbf{return},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa}\rangle \stackrel{return}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}',\ \mathsf{os}',\ \mathsf{fp}'\rangle,\\ \textbf{if}\ \mathsf{fp} \in \mathsf{framePointer}_{\mathsf{range}};\end{array}}\ [\mathsf{return}]$$

The instruction **areturn** is identical to **sreturn** and thus not shown here.

4.19 Object operations

The new operation creates an instance of the class identified by the given class index ci. The class index is used to lookup the class in the class table pertaining to the current application program, which itself is found by using the current application program id pi as an index in the application program table. The fields are initialised to zeroes.

$$\frac{\begin{array}{l}\vdash \mathsf{pt}(\mathsf{pi}) \stackrel{ct}{\Rightarrow} \mathsf{ct},\\ \vdash \mathsf{ct}(\mathsf{ci}) \stackrel{ob}{\Rightarrow} \textbf{ClassObject}\ \mathsf{oh}\ \mathsf{is}\ \_\ \_\ \_\ \_,\\ \vdash \{\mathsf{i} \mapsto 0 \mid \mathsf{i} \leftarrow [0..\mathsf{is}-1]\} \stackrel{ft}{\Rightarrow} \mathsf{ft},\\ \vdash \mathsf{hp}+1 \stackrel{hp}{\Rightarrow} \mathsf{hp}',\\ \vdash \mathsf{s2p}(\mathsf{hp}') \stackrel{p}{\Rightarrow} (\mathsf{hi_r},\ \mathsf{lo_r}),\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{hi_r}\} \oplus \{\mathsf{sp}+2 \mapsto \mathsf{lo_r}\} \stackrel{os}{\Rightarrow} \mathsf{os}',\\ \vdash \mathsf{ha} \oplus \{\mathsf{hp}' \mapsto \textbf{RegularObject}\ \mathsf{oh}\ \mathsf{ft}\} \stackrel{ha}{\Rightarrow} \mathsf{ha}'\end{array}}{\begin{array}{l}\vdash \langle \textbf{new}\ \mathsf{ci},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt}\rangle \stackrel{object}{\Rightarrow} \langle \mathsf{sp}+2,\ \mathsf{os}',\ \mathsf{hp}',\ \mathsf{ha}'\rangle,\\ \textbf{if}\ (\mathsf{sp}+1 \ldots \mathsf{sp}+2) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge\\ \ \mathsf{pi} \in \mathsf{progId}_{\mathsf{range}} \wedge \mathsf{ci} \in \mathsf{classId}_{\mathsf{range}} \wedge\\ \ \mathsf{hp}' \in \mathsf{heapPointer}_{\mathsf{range}};\end{array}}\ [\mathsf{new}]$$

There are three instructions to determine whether an object is an instance of a particular class. The **instanceof** instruction is for regular objects. The two other instructions **ainstanceof** and **aainstanceof** handle array objects of primitive, respectively, non-primitive types.

The immediate operand $\mathsf{ci_t}$ of the instruction **instanceof** must be the index into the class table of some regular class, t say. In addition, the top of the stack must contain a reference r to a regular object of some class, s say. If t and s are the same, or if t is a super class of s, the instruction pushes 1 on the operand stack; 0 otherwise. (See Figure 5 for the definition of b2b).

$$\frac{\begin{array}{l}\vdash \mathsf{p2s(os(sp-1),\ os(sp))} \overset{s}{\Rightarrow} \mathsf{r},\\ \vdash \mathsf{ha(r)} \overset{ob}{\Rightarrow} \textbf{RegularObject}(\textbf{ObjectHeader}\ \_\ \_(\textbf{MethodTable}\ \mathsf{ci_s}\ \_))\_,\\ \vdash \mathsf{pt(pi)} \overset{ct}{\Rightarrow} \mathsf{ct},\\ \vdash \mathsf{ct(ci_s)} \overset{ob}{\Rightarrow} \textbf{ClassObject}\ \_\ \_\ \_\ \_\ \mathsf{super}\ \_,\\ \vdash \mathsf{b2b(ci_t = ci_s \vee ci_t \in range(super))} \overset{b}{\Rightarrow} \mathsf{v},\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{v}\} \overset{os}{\Rightarrow} \mathsf{os'}\end{array}}{\begin{array}{l}\vdash \langle \textbf{instanceof}\ \mathsf{ci_t,\ sp,\ os,\ hp,\ ha,\ pi,\ pt}\rangle \overset{instance}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{os'}\rangle,\\ \textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer_{range}} \wedge\\ \ \mathsf{r} \in \mathsf{heapPointer_{range}} \wedge \mathsf{isRegularObject(ha(r))} \wedge\\ \ \mathsf{pi} \in \mathsf{progId_{range}} \wedge \mathsf{ci_s} \in \mathsf{classId_{range}};\end{array}}\ [\mathsf{instanceof}]$$

The immediate operand $\mathsf{dt_t}$ of the instruction **ainstanceof** must specify one of the three primitive array types, t say. The top of the stack r must point at an array of primitive types, s say. If t and s are the same, 1 is pushed on the operand stack; 0 otherwise.

$$\frac{\begin{array}{l}\vdash \mathsf{p2s(os(sp-1),\ os(sp))} \overset{s}{\Rightarrow} \mathsf{r},\\ \vdash \mathsf{ha(r)} \overset{ob}{\Rightarrow} \textbf{ArrayObject}(\textbf{ArrayHeader}\ \_\ \_\ \_\ \mathsf{dt_s}\ \_)\_,\\ \vdash \mathsf{b2b(dt_s \in \{\textbf{bit},\ \textbf{byte},\ \textbf{short}\} \wedge dt_t = dt_s)} \overset{b}{\Rightarrow} \mathsf{v},\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{v}\} \overset{os}{\Rightarrow} \mathsf{os'}\end{array}}{\begin{array}{l}\vdash \langle \textbf{ainstanceof}\ \mathsf{dt_t,\ sp,\ os,\ hp,\ ha,\ pi,\ pt}\rangle \overset{instance}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{os'}\rangle,\\ \textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer_{range}} \wedge\\ \ \mathsf{r} \in \mathsf{heapPointer_{range}} \wedge \mathsf{isArrayObject(ha(r))};\end{array}}\ [\mathsf{ainstanceof}]$$

The immediate operand $\mathsf{ci_t}$ of the instruction **ainstanceof** must be the index into the class table of some regular class, t say. The top of the stack must contain a reference r to an array object, whose elements are instances of some class, s say. If t and s are the same, or if t is a super class of s, the instruction pushes 1 on the operand stack; 0 otherwise.

$$\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{s}{\Rightarrow} \mathsf{r},\\ \vdash \mathsf{ha}(\mathsf{r}) \stackrel{ob}{\Rightarrow} \textbf{ArrayObject}(\textbf{ArrayHeader}\ \_\ \mathsf{ci_s}\ \_\ \textbf{ref}\ \_)\_,\\ \vdash \mathsf{pt}(\mathsf{pi}) \stackrel{ct}{\Rightarrow} \mathsf{ct},\\ \vdash \mathsf{ct}(\mathsf{ci_s}) \stackrel{ob}{\Rightarrow} \textbf{ClassObject}\ \_\ \_\ \_\ \_\ \mathsf{super}\ \_,\\ \vdash \mathsf{b2b}(\mathsf{ci_t} = \mathsf{ci_s} \vee \mathsf{ci_t} \in \mathsf{range}(\mathsf{super})) \stackrel{b}{\Rightarrow} \mathsf{v},\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{v}\} \stackrel{os}{\Rightarrow} \mathsf{os}'\end{array}}{\vdash \langle \textbf{aainstanceof}\ \mathsf{ci_t},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt}\rangle \stackrel{instance}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{os}'\rangle,}\ [\mathsf{aainstanceof}]$$
$$\begin{array}{l}\textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer_{range}} \wedge\\ \ \mathsf{r} \in \mathsf{heapPointer_{range}} \wedge \mathsf{isArrayObject}(\mathsf{ha}(\mathsf{r})) \wedge\\ \ \mathsf{pi} \in \mathsf{progId_{range}} \wedge \mathsf{ci_s} \in \mathsf{classId_{range}};\end{array}$$

The three instructions **checkcast**, **acheckcast** and **aacheckcast** below handle, regular objects, array objects of primitive, respectively, non-primitive types in the same way as the three 'instance of' instructions above.

The **checkcast** instruction permits a null reference to be cast to any other reference. Otherwise **instanceof** is used to determine whether the cast is acceptable. The operand stack is unaffected.

$$\frac{\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{s}{\Rightarrow} \mathsf{r}}{\vdash \langle \textbf{checkcast}\ \mathsf{ci},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt}\rangle \stackrel{instance}{\Rightarrow} \langle \mathsf{sp},\ \mathsf{os}\rangle,}\ [\mathsf{checkcast}^0]$$
$$\textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer_{range}} \wedge \mathsf{r} = \mathsf{nullreference};$$

$$\frac{\begin{array}{l}\vdash \langle \textbf{instanceof}\ \mathsf{ci},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt}\rangle \stackrel{instance}{\Rightarrow} \langle \mathsf{sp}',\ \mathsf{os}'\rangle,\\ \vdash \mathsf{os}'(\mathsf{sp}') \stackrel{b}{\Rightarrow} \mathsf{v}\end{array}}{\vdash \langle \textbf{checkcast}\ \mathsf{ci},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt}\rangle \stackrel{instance}{\Rightarrow} \langle \mathsf{sp},\ \mathsf{os}\rangle,}\ [\mathsf{checkcast}^1]$$
$$\textbf{if}\ \mathsf{v} = 1;$$

The two instructions **acheckcast** and **aacheckcast** rely on the appropriate 'instance of' instructions in a similar way. They are not shown here.

4.20 Loading and storing object fields

The two 'get' instructions below load a value from an object field onto the operand stack. The two 'put' instructions serve to store a field with a byte or a short. There are no agetfield or aputfield instruction. The side conditions check for stack underflow, null references, or a reference to an object of the wrong type. Illegal field indices should be detected by the static semantics.

$$\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{s}{\Rightarrow} \mathsf{r},\\ \vdash \mathsf{ha}(\mathsf{r}) \stackrel{ob}{\Rightarrow} \textbf{RegularObject}\ \mathsf{oh}\ \mathsf{ft},\\ \vdash \mathsf{ft}(\mathsf{i}) \stackrel{b}{\Rightarrow} \mathsf{v}\end{array}}{\vdash \langle \textbf{bgetfield}\ \mathsf{i},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{ha}\rangle \stackrel{getfield}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{v}\}\rangle,}\ [\mathsf{bgetfield}]$$
$$\begin{array}{l}\textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer_{range}} \wedge\\ \ \mathsf{r} \in \mathsf{heapPointer_{range}} \wedge \mathsf{isRegularObject}(\mathsf{ha}(\mathsf{r}));\end{array}$$

$$\dfrac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{s}{\Rightarrow} \mathsf{r},\\ \vdash \mathsf{ha}(\mathsf{r}) \overset{ob}{\Rightarrow} \textbf{RegularObject}\ \mathsf{oh}\ \mathsf{ft},\\ \vdash (\mathsf{ft}(\mathsf{i}),\ \mathsf{ft}(\mathsf{i}+1)) \overset{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}-1 \mapsto \mathsf{hi}\} \oplus \{\mathsf{sp} \mapsto \mathsf{lo}\} \overset{os}{\Rightarrow} \mathsf{os}'\end{array}}{\vdash \langle \textbf{sgetfield}\ \mathsf{i},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{ha}\rangle \overset{getfield}{\Rightarrow} \langle \mathsf{sp},\ \mathsf{os}'\rangle,}\ [\mathsf{sgetfield}]$$
$$\textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge \mathsf{r} \in \mathsf{heapPointer}_{\mathsf{range}} \wedge \mathsf{isRegularObject}(\mathsf{ha}(\mathsf{r}));$$

$$\dfrac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-2),\ \mathsf{os}(\mathsf{sp}-1)) \overset{s}{\Rightarrow} \mathsf{r},\\ \vdash \mathsf{os}(\mathsf{sp}) \overset{b}{\Rightarrow} \mathsf{v},\\ \vdash \mathsf{ha}(\mathsf{r}) \overset{ob}{\Rightarrow} \textbf{RegularObject}\ \mathsf{oh}\ \mathsf{ft},\\ \vdash \mathsf{ft} \oplus \{\mathsf{i} \mapsto \mathsf{v}\} \overset{ft}{\Rightarrow} \mathsf{ft}',\\ \vdash \mathsf{ha} \oplus \{\mathsf{r} \mapsto \textbf{RegularObject}\ \mathsf{oh}\ \mathsf{ft}'\} \overset{ha}{\Rightarrow} \mathsf{ha}'\end{array}}{\vdash \langle \textbf{bputfield}\ \mathsf{i},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{ha}\rangle \overset{putfield}{\Rightarrow} \langle \mathsf{sp}-3,\ \mathsf{ha}'\rangle,}\ [\mathsf{bputfield}]$$
$$\textbf{if}\ (\mathsf{sp}-2 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge \mathsf{r} \in \mathsf{heapPointer}_{\mathsf{range}} \wedge \mathsf{isRegularObject}(\mathsf{ha}(\mathsf{r}));$$

$$\dfrac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-3),\ \mathsf{os}(\mathsf{sp}-2)) \overset{s}{\Rightarrow} \mathsf{r},\\ \vdash (\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \overset{p}{\Rightarrow} (\mathsf{hi},\ \mathsf{lo}),\\ \vdash \mathsf{ha}(\mathsf{r}) \overset{ob}{\Rightarrow} \textbf{RegularObject}\ \mathsf{oh}\ \mathsf{ft},\\ \vdash \mathsf{ft} \oplus \{\mathsf{i} \mapsto \mathsf{hi}\} \oplus \{\mathsf{i}+1 \mapsto \mathsf{lo}\} \overset{ft}{\Rightarrow} \mathsf{ft}',\\ \vdash \mathsf{ha} \oplus \{\mathsf{r} \mapsto \textbf{RegularObject}\ \mathsf{oh}\ \mathsf{ft}'\} \overset{ha}{\Rightarrow} \mathsf{ha}'\end{array}}{\vdash \langle \textbf{sputfield}\ \mathsf{i},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{ha}\rangle \overset{putfield}{\Rightarrow} \langle \mathsf{sp}-4,\ \mathsf{ha}'\rangle,}\ [\mathsf{sputfield}]$$
$$\textbf{if}\ (\mathsf{sp}-3 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer}_{\mathsf{range}} \wedge \mathsf{r} \in \mathsf{heapPointer}_{\mathsf{range}} \wedge \mathsf{isRegularObject}(\mathsf{ha}(\mathsf{r}));$$

4.21 Loading and storing static objects

Static objects are kept in the static area. The instructions **bgetstatic**, **sgetstatic**, **bputstatic**, and **sputstatic** are used to manipulate static objects.

$$\dfrac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{hi_r},\ \mathsf{lo_r}) \overset{s}{\Rightarrow} \mathsf{i},\\ \vdash \mathsf{sa}(\mathsf{i}) \overset{b}{\Rightarrow} \mathsf{v},\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{v}\} \overset{os}{\Rightarrow} \mathsf{os}'\end{array}}{\vdash \langle \textbf{bgetstatic}\ \mathsf{hi_r}\ \mathsf{lo_r},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{sa}\rangle \overset{getstatic}{\Rightarrow} \langle \mathsf{sp}+1,\ \mathsf{os}'\rangle,}\ [\mathsf{bgetstatic}]$$
$$\textbf{if}\ (\mathsf{sp}+1) \in \mathsf{stackPointer}_{\mathsf{range}};$$

$$\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{hi_r},\ \mathsf{lo_r}) \stackrel{s}{\Rightarrow} \mathsf{i},\\ \vdash(\mathsf{sa}(\mathsf{i}),\ \mathsf{sa}(\mathsf{i}+1)) \stackrel{p}{\Rightarrow} (\mathsf{hi_v},\ \mathsf{lo_v}),\\ \vdash \mathsf{os} \oplus \{\mathsf{sp}+1 \mapsto \mathsf{hi_v}\} \oplus \{\mathsf{sp}+2 \mapsto \mathsf{lo_v}\} \stackrel{os}{\Rightarrow} \mathsf{os'}\end{array}}{\vdash\langle \textbf{sgetstatic}\ \mathsf{hi_r}\ \mathsf{lo_r},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{sa}\rangle \stackrel{getstatic}{\Rightarrow} \langle \mathsf{sp}+2,\ \mathsf{os'}\rangle,}\ [\mathsf{sgetstatic}]$$
$\textbf{if}\ (\mathsf{sp}+1 \ldots \mathsf{sp}+2) \subseteq \mathsf{stackPointer_{range}};$

$$\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{hi_r},\ \mathsf{lo_r}) \stackrel{s}{\Rightarrow} \mathsf{i},\\ \vdash \mathsf{os}(\mathsf{sp}) \stackrel{b}{\Rightarrow} \mathsf{v}\end{array}}{\vdash\langle \textbf{bputstatic}\ \mathsf{hi_r}\ \mathsf{lo_r},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{sa}\rangle \stackrel{putstatic}{\Rightarrow} \langle \mathsf{sp}-1,\ \mathsf{sa} \oplus \{\mathsf{i} \mapsto \mathsf{v}\}\rangle,}\ [\mathsf{bputstatic}]$$
$\textbf{if}\ \mathsf{sp} \in \mathsf{stackPointer_{range}};$

$$\frac{\begin{array}{l}\vdash \mathsf{p2s}(\mathsf{hi_r},\ \mathsf{lo_r}) \stackrel{s}{\Rightarrow} \mathsf{i},\\ \vdash(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{p}{\Rightarrow} (\mathsf{hi_v},\ \mathsf{lo_v}),\\ \vdash \mathsf{sa} \oplus \{\mathsf{i} \mapsto \mathsf{hi_v}\} \oplus \{\mathsf{i}+1 \mapsto \mathsf{lo_v}\} \stackrel{sa}{\Rightarrow} \mathsf{sa'}\end{array}}{\vdash\langle \textbf{sputstatic}\ \mathsf{hi_r}\ \mathsf{lo_r},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{sa}\rangle \stackrel{putstatic}{\Rightarrow} \langle \mathsf{sp}-2,\ \mathsf{sa'}\rangle,}\ [\mathsf{sputstatic}]$$
$\textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer_{range}};$

4.22 Miscellaneous instructions

The **breakpoint** instruction pops the top two elements of the operand stack, interprets them as the high and low byte of a short and appends the short to the output stream.

$\mathsf{outputStream} \equiv [\mathsf{short}];$

$$\frac{\vdash \mathsf{p2s}(\mathsf{os}(\mathsf{sp}-1),\ \mathsf{os}(\mathsf{sp})) \stackrel{s}{\Rightarrow} \mathsf{v}}{\vdash\langle \textbf{breakpoint},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{output}\rangle \stackrel{breakpoint}{\Rightarrow} \langle \mathsf{sp}-2,\ \mathsf{output} +\!\!+ [\mathsf{v}]\rangle,}\ [\mathsf{breakpoint}]$$
$\textbf{if}\ (\mathsf{sp}-1 \ldots \mathsf{sp}) \subseteq \mathsf{stackPointer_{range}};$

4.23 Combining the rules

The semantics of the 25 subsets of the instruction set are specified by as many different relations, such as $\stackrel{const}{\Rightarrow}$. These different relations are embedded in the relation $\stackrel{exec}{\Rightarrow}$ by the rules below. The $\stackrel{exec}{\Rightarrow}$ relation also automatically increments the program counter by one upon completing the execution of an instruction, with a few exceptions detailed below.

The separation of the different categories of instructions shows that the specification is modular: The configuration of the virtual machine has 12 components, which is quite large. However, the relation for many of the subsets uses only a small number of components, thus hiding the remaining components.

$$\frac{\vdash\langle \mathsf{constInst},\ \mathsf{sp},\ \mathsf{os}\rangle \stackrel{const}{\Rightarrow} \langle \mathsf{sp'},\ \mathsf{os'}\rangle}{\vdash\langle \mathsf{pc},\ \mathsf{ca},\ \mathsf{constInst},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt},\ \mathsf{sa},\ \mathsf{output}\rangle}\ [\mathsf{exec^{const}}]$$
$\stackrel{exec}{\Rightarrow} \langle \mathsf{pc}+1,\ \mathsf{sp'},\ \mathsf{os'},\ \mathsf{fp},\ \mathsf{fa},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{sa},\ \mathsf{output}\rangle;$

Most other relations defining subsets of the instruction set are embedded in the relation $\overset{execs}{\Rightarrow}$ in the same way as shown above. The exception to this rule is formed by the relations $\overset{return}{\Rightarrow}$, $\overset{control}{\Rightarrow}$, $\overset{switch}{\Rightarrow}$, and $\overset{invoke\ldots}{\Rightarrow}$, which calculate the new value of the program counter pc'. The automatic increment of the program counter is thus suppressed.

$$[\mathsf{exec}^{\mathsf{return}}]\ \frac{\vdash\langle \mathsf{returnInst},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa}\rangle \overset{return}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}',\ \mathsf{os}',\ \mathsf{fp}'\rangle}{\vdash\langle \mathsf{pc},\ \mathsf{ca},\ \mathsf{returnInst},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt},\ \mathsf{sa},\ \mathsf{output}\rangle \overset{exec}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}',\ \mathsf{os}',\ \mathsf{fp}',\ \mathsf{fa},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{sa},\ \mathsf{output}\rangle};$$

4.24 Main semantic function

The function jsp defines the semantics of a JSP programas the transitive closure of the relation $\overset{decode}{\Rightarrow}$ (below). When given an initial JSP machine configuration, jsp computes a list of successive configurations that can be inspected.

$$\begin{array}{ll}\mathsf{configuration} & \equiv \langle \mathsf{programCounter},\ \mathsf{codeArea},\ \mathsf{stackPointer},\ \mathsf{operandStack},\\ & \quad \mathsf{framePointer},\ \mathsf{frameArea},\ \mathsf{heapPointer},\ \mathsf{heapArea},\\ & \quad \mathsf{progId},\ \mathsf{progTable},\ \mathsf{staticArea},\ \mathsf{outputStream}\rangle;\\ \mathsf{jsp} & :: \mathsf{configuration} \rightarrow [\mathsf{configuration}];\\ \mathsf{jsp\ s0} & = (\mathsf{s0} \overset{decode}{\Rightarrow} *);\end{array}$$

The relation $\overset{decode}{\Rightarrow}$ accesses the instruction at the current program counter. The case analysis by the $\overset{exec}{\Rightarrow}$ relation decides to which category the current instruction belongss and delegates the actual processing of the instruction to the appropriate embedded relation.

$$\overset{decode}{\Rightarrow} :: (\mathsf{configuration} \leftrightarrow \mathsf{configuration});$$

$$[\mathsf{decode}]\ \frac{\vdash\langle \mathsf{pc},\ \mathsf{ca},\ \mathsf{ca(pc)},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt},\ \mathsf{sa},\ \mathsf{output}\rangle \overset{exec}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{sp}',\ \mathsf{os}',\ \mathsf{fp}',\ \mathsf{fa}',\ \mathsf{hp}',\ \mathsf{ha}',\ \mathsf{sa}',\ \mathsf{output}'\rangle}{\vdash\langle \mathsf{pc},\ \mathsf{ca},\ \mathsf{sp},\ \mathsf{os},\ \mathsf{fp},\ \mathsf{fa},\ \mathsf{hp},\ \mathsf{ha},\ \mathsf{pi},\ \mathsf{pt},\ \mathsf{sa},\ \mathsf{output}\rangle \overset{decode}{\Rightarrow} \langle \mathsf{pc}',\ \mathsf{ca},\ \mathsf{sp}',\ \mathsf{os}',\ \mathsf{fp}',\ \mathsf{fa}',\ \mathsf{hp}',\ \mathsf{ha}',\ \mathsf{pi},\ \mathsf{pt},\ \mathsf{sa}',\ \mathsf{output}'\rangle};$$

A sample machine configuration such as test (see Section 6) can be supplied as an argument to jsp.

5 On the relationship between the JVM and the JSP

The JSP is essentially a scaled down version of the JVM. However, the JSP byte codes are not a strict subset of the JVM and translating JVM byte codes into JSP byte codes presents some interesting problems. This section comments on the relationship between the two virtual machines and sketches a simplified process of translating Java class files into the tables required to run JSP code.

The main problem of translating JVM byte codes into JSP bytecodes is the pervasive use of 32-bit data in Java programs. The translator built by Java Soft performs a sophisticated analysis to ensure that the computations performed by the JSP have the same semantics as those carried out by the JVM. The results of the analysis enable the translator to map certain integers and associated operations on bytes, and some on shorts. The translator also inserts instructions to support multiple precision arithmetic when genuine 32-bit integers are needed.

The simplified translation to be described here assumes that all integers can be represented as shorts. We make no attempt to either identify opportunities for using bytes or to warn if shorts are too limited.

The translation of Java class files into the tables required by the JSP consists of the following steps:

- To allocate all statics in the staticArea, to create an index of all application programs in the progTable, and to gather the code sections of all methods in the codeArea.
- For each application program to allocate a classTable.
- For each class to allocate a classObject with its objectHeader, a methodTable, a superTable, and an interfaceTable, and to decide on the layout of the fields in the instance of the class.
- For each method to allocate a methodHeader, to gather the byte codes of the method and to decide on a start address of the method.
- For each word offset, address or integer to convert it into a short. Depending on the sophistication of the translation process this may simply truncate all values, or restructure the byte code to deal with values that cannot be fit into 16 bits.
- For each instruction to convert it as indicated below.

To present the translation of individual JVM byte codes into JSP bytes codes in a reasonably succinct manner we use the following abbreviations.

- byte, short, index, params and address stand for numeric values in the appropriate range.
- class, field, method, and static stand for the appropriate name.
- $[a|b|c]$ stands for exactly one of the words a, b or c.

We list all JVM instructions [7] (on the lft), and describe the equivalent JSP instruction or sequence of instructions (on the right).

- Constant instructions.
  nop = **nop**;
  bipush byte = **spush** 0 byte;
  sipush short = **spush**(short div 256)(short mod 256);
  $\mathsf{aconst}_{\mathsf{null}}$ = $\mathbf{aconst}_{\mathsf{null}}$;
  $\mathsf{iconst}_{\mathsf{m1}}$ = $\mathbf{bconst}_{0}$, $\mathbf{bconst}_{\mathsf{m1}}$;
  $\mathsf{iconst}_{[0|1|2|3|4|5]}$ = $\mathbf{bconst}_{0}$, $\mathbf{bconst}_{[0|1|2|3|4|5]}$;
  iconst short = **bpush**(short div 256), **bpush**(short mod 256);
  iconst byte = **bpush** byte;
- The load, store and increment instructions.

[a|i]load$_{[0|1]}$ = **[A|S]load$_{[0|2]}$**;
[a|i]load$_{[2|3]}$ = **[A|S]load** [4|6];
[a|i]load index = **[A|S]load**(2*index);
[a|i]store$_{[0|1]}$ = **[A|S]store$_{[0|2]}$**;
[a|i]store$_{[2|3]}$ = **[A|S]store** [4|6];
[a|i]store index = **[A|S]store**(2*index);
iinc index byte = **sinc**(2*index)byte;

– Stack instructions.
dup = **dup2**;
dup_x [1|2] = **dup_x**(2*16 + [2|4]);
dup2 = **dup_x**(4*16 + 4);
dup2_x [1|2] = **dup_x**(4*16 + [6|8]);
pop = **pop2**;
pop2 = **pop2**, **pop2**;
swap = **swap2**;

– Array creation, load and store instructions.
anewarray class = **anewarray** class;
newarray [boolean|byte|short|int] = **newarray** [**bit**|**byte**|**short**|**short**];
arraylength = **arraylength**;
[a|b|i|s]load = **[A|B|S|S]load**;
[a|b|i|s]store = **[A|B|S|S]store**;

– Instructions for arithmetical, logical and conversion operations.
i[add|sub|mul|div|rem] = **S[add|sub|mul|div|rem]**;
i[shl|shr|ushr] = **S[shl|shr|ushr]**;
i[and|or|xor] = **S[and|or|xor]**;
i2b = **s2b**;
i2s = **nop**;

– The JVM Conditional branches translate into a number of JSP instructions.
ifnonnull address = **aconst$_{\text{null}}$**, **acmp**, **ifne** address;
ifnull address = **aconst$_{\text{null}}$**, **acmp**, **ifeq** address;
if[a|i]cmp[eq|lt|gt|ne|ge|le] address = **[A|S]cmp**, **If[eq|lt|gt|ne|ge|le]** address;
if[eq|lt|gt|ne|ge|le] address = **s2b**, **If[eq|lt|gt|ne|ge|le]** address;
goto address = **goto** address;

– The JVM instructions tableswitch and lookupswitch are variable length instructions. The tables may contain an arbitrary number of index/target or key/target pairs.
tableswitch from to default{index $\mapsto$ address} =
tableswitch default from to{index $\mapsto$ address};
lookupswitch size default{index $\mapsto$ (key, address)} =
lookupswitch default size{index $\mapsto$ (key, address)};

– Exception handling.
athrow = **athrow**;
jsr address = **jsr** address;
ret index = **ret**(2*index);

– Instructions for method invokation.

invokeinterface params class method = **invokeinterface** params class method;
invokespecial address = **invoke** address;
invokestatic address = **invoke** address;
invokevirtual params method = **invokevirtual** params method;
[a|i]return = **[A|S]return**;
return = **return**;

– Instructions for object creation and manipulation.
  new class = **new** class;
  instanceof class = **instanceof** class, **b2s**;
  checkcast class = **checkcast** class;
  getfield field = **sgetfield** field;
  putfield field = **sputfield** field;
  getstatic static = **sgetstatic** static;
  putstatic static = **sputstatic** static;
– Miscellaneous instructions.
  breakpoint = **breakpoint**;
– All other JVM instructions are unsupported. These are jsr_w, goto_w, wide, monitorenter, monitorexit, multianewarray, and all instructions involving character, long, float, and double data types.

We use SUN's Java compiler from the Java Development Kit version 1.1 to generate class files from sample Java programs. The translations sketched above have been implemented as a simple sed/awk script, such that the results of the translation can be used as sample input for the main semantic function jsp. This will be explored briefly in the next section.

6 A sample program

We have written a suite of simple Java programs, varying from quick sort to specific tests for the object system, to validate aspects of the semantics. The workings of the JSP semantics is best illustrated by exposing some details of a representative program from our suite. The program below is a slightly modified version of [4, Page 48]. The two calls to `println` have been added to show that the program is working. Furthermore we have added the call to `setColor` to demonstrate the workings of multiple inheritance.

```
public class Point{ int x, y; } ;

public interface Colorable {
  void setColor( byte r, byte g, byte b) ;
}

public class ColoredPoint extends Point implements Colorable {
  byte r,g,b;
  public void setColor( byte rv, byte gv, byte bv ) {
    r = rv ; g = gv; b = bv ;
```

```
  }
}

public class test {
  public static void main( String [] args ) {
    Point p = new Point() ;
    ColoredPoint cp = new ColoredPoint() ;
    p = cp ;
    System.out.println( p.x ) ;
    Colorable c = cp ;
    c.setColor( (byte) 0, (byte) 1, (byte) 2 ) ;
    System.out.println( cp.b ) ;
  }
}
```

The 12 components of the JSP virtual machine configuration necessary to execute $\mathsf{test.main}$ are initialised as follows:

program counter The program counter is initialised to 0.

code area The code for all methods to be executed by the current application program (which includes the initialiser for $\mathsf{java.lang.Object}$) is gathered in the code area. An extra instruction at address zero is added to the code area whose task it is to invoke the main method. This is represented as $0 \mapsto (\textbf{invoke}\ \mathsf{s2p}(\mathsf{test.main}_{\mathsf{pc}}))$

stack pointer The initial value of the stack pointer is argc.
$\mathsf{argc} :: \mathsf{stackPointer};$
$\mathsf{argc} = 1;$

operand stack Initially the operand stack is the same as argv.
$\mathsf{argv} :: \mathsf{operandStack};$
$\mathsf{argv} = \{0 \mapsto 0,\ 1 \mapsto 0\};$

frame pointer The initial value of the frame pointer is -1, to indicate that the frame area is initially empty.

frame area The initial frame area is empty.

heap pointer The initial heap pointer is -1, indicating an empty heap.

heap The heap is initially empty.

application program index $\mathsf{test}_{\mathsf{pi}}$ is the index in the application program table of the current application program. The formal specification presently does not specify a mechanism for switching application programs.

application program table $\mathsf{machine}_{\mathsf{pt}}$ is the machine wide mapping from application program ids to a class tables, providing one class table per application program.

Static area $\mathsf{machine}_{\mathsf{sa}}$ is the machine wide area used to store static values. The sample program does not have any static values.

Initial output The initial output stream is empty.

$$\begin{aligned}
&\mathsf{test} :: \mathsf{configuration};\\
&\mathsf{test} = \langle 0,\ \{0 \mapsto \textbf{invoke}(\mathsf{s2p}(\mathsf{test.main}_{\mathsf{pc}}))\} \cup \mathsf{machine}_{\mathsf{ca}},\\
&\qquad \mathsf{argc},\ \mathsf{argv},\ -1,\ \{\},\ -1,\ \{\},\ \mathsf{test}_{\mathsf{pi}},\ \mathsf{machine}_{\mathsf{pt}},\ \mathsf{machine}_{\mathsf{sa}},\ []\rangle;
\end{aligned}$$

The JSP byte codes for the main method of class test are shown below. Instead of calling the println method of the library class System, we use the **breakpoint** instruction to inspect the configuration of the machine.

$$
\begin{array}{l}
\mathsf{test.main_{ca}} :: \mathsf{codeArea};\\
\mathsf{test.main_{ca}} = \{\mathsf{test.main_{pc}} - 1 \mapsto \textbf{MethodHeader}\ \mathsf{False}\ \mathsf{False}\ 8\ 2\ 6,\\
\qquad \mathsf{test.main_{pc}} + 0 \mapsto \textbf{new}\ \mathsf{Point_{ci}},\\
\qquad \mathsf{test.main_{pc}} + 1 \mapsto \textbf{dup2},\\
\qquad \mathsf{test.main_{pc}} + 2 \mapsto \textbf{invoke}(\mathsf{s2p}\ \mathsf{Point.init_{pc}}),\\
\qquad \mathsf{test.main_{pc}} + 3 \mapsto \textbf{astore}_2,\\
\qquad \mathsf{test.main_{pc}} + 4 \mapsto \textbf{new}\ \mathsf{ColoredPoint_{ci}},\\
\qquad \mathsf{test.main_{pc}} + 5 \mapsto \textbf{dup2},\\
\qquad \mathsf{test.main_{pc}} + 6 \mapsto \textbf{invoke}(\mathsf{s2p}\ \mathsf{ColoredPoint.init_{pc}}),\\
\qquad \mathsf{test.main_{pc}} + 7 \mapsto \textbf{astore}\ 4,\\
\qquad \mathsf{test.main_{pc}} + 8 \mapsto \textbf{aload}\ 4,\\
\qquad \mathsf{test.main_{pc}} + 9 \mapsto \textbf{astore}_2,\\
\qquad \mathsf{test.main_{pc}} + 10 \mapsto \textbf{nop},\\
\qquad \mathsf{test.main_{pc}} + 11 \mapsto \textbf{aload}_2,\\
\qquad \mathsf{test.main_{pc}} + 12 \mapsto \textbf{sgetfield}\ \mathsf{Point.x_{fi}},\\
\qquad \mathsf{test.main_{pc}} + 13 \mapsto \textbf{breakpoint},\\
\qquad \mathsf{test.main_{pc}} + 14 \mapsto \textbf{aload}\ 4,\\
\qquad \mathsf{test.main_{pc}} + 15 \mapsto \textbf{astore}\ 6,\\
\qquad \mathsf{test.main_{pc}} + 16 \mapsto \textbf{aload}\ 6,\\
\qquad \mathsf{test.main_{pc}} + 17 \mapsto \textbf{bconst}_0,\ \mathsf{test.main_{pc}} + 18 \mapsto \textbf{bconst}_0,\\
\qquad \mathsf{test.main_{pc}} + 19 \mapsto \textbf{bconst}_0,\ \mathsf{test.main_{pc}} + 20 \mapsto \textbf{bconst}_1,\\
\qquad \mathsf{test.main_{pc}} + 21 \mapsto \textbf{bconst}_0,\ \mathsf{test.main_{pc}} + 22 \mapsto \textbf{bconst}_2,\\
\qquad \mathsf{test.main_{pc}} + 23 \mapsto \textbf{invokeinterface}\ 8\ \mathsf{Colorable_{ii}}\ \mathsf{Colorable.setColor_{mi}},\\
\qquad \mathsf{test.main_{pc}} + 24 \mapsto \textbf{nop},\\
\qquad \mathsf{test.main_{pc}} + 25 \mapsto \textbf{aload}\ 4,\\
\qquad \mathsf{test.main_{pc}} + 26 \mapsto \textbf{sgetfield}\ \mathsf{ColoredPoint.b_{fi}},\\
\qquad \mathsf{test.main_{pc}} + 27 \mapsto \textbf{breakpoint},\\
\qquad \mathsf{test.main_{pc}} + 28 \mapsto \textbf{return}\};
\end{array}
$$

The execution of the program can be expressed simply as jsp(test). The `latos` tool makes it possible to trace the execution of the program, and to experiment with different initial configurations.

The program starts by creating two heap objects, one representing a Point and the second representing a ColoredPoint. The objects are properly initialised by a chain of calls to the initialisers of the super classes. The most interesting instruction is the **invokeinterface**, which has to discover that the instance of ColoredPoint indeed implements the setColor method.

The program causes two values to be appended to the output stream (via the **breakpoint** instruction). The values are 0 (because the coordinates of the class Point are initialised to 0) and 2 (because ColoredPoint.setColor assigns this value to the field cp.b).

7 Conclusions and future work

The result of formalising the operational semantics of the JSP is a specification that is:

- succinct, because it is shorter and more detailed than the natural language documents.
- clear, because the rules are not open to more than one interpretation.
- executable, because a program can be generated automatically from the specification, which can subsequently be executed to validate and explore the behaviour of sample Java programs.
- consistent, because the tools available for the notation used check well formedness, types and source dependency.
- modular, because sub sets of rules can be considered in isolation.
- large, because it has to cope with 25 groups of in total 124 different JSP instructions.
- not difficult to read, because the rules describing the semantics of many instructions are similar.

The fact that our specification is executable allows implementors to experiment with Java programs and byte codes, inspect the configuration of the JSP and generally sharpen their understanding of the mechanisms. Without tool support it would be impossible to construct a derivation tree for anything but the most trivial Java programs. With the help of our `latos` tool, our specification could be used to automatically construct derivation trees for small to medium sized programs.

We hope to be able to make our complete specification available on the Web, so that others may down load the specification and the `latos` tool and use these resources whilst implementing a JSP.

In future we hope to gain access to a complete operational semantics of the JVM, formally specify the JVM to JSP translator and attempt to give a correctness proof of the translator with respect to the semantics of the JVM byte codes and that of the JSP byte codes.

We have not considered the static semantics of a JSP, that is a specification of properties of JSP programs that can be be checked statically, for example by the JVM to JSP byte code translator, or the byte code verifier. An important goal would be to investigate which static properties of the JVM that are preserved by the JVM to JSP translator. The work of Stata and Abadi [10] offers a promising basis for this.

References

1. ISO/IEC 7816-4:1995. *Information technology–Identification cards–Integrated circuit(s) cards with contacts part4: Inter-Industry commands for interchange.* International Standards Organization, 1995.

2. P. Bertelsen. Semantics of Java byte code. Technical report, Technical Univ. of Denmark, Mar 1997. `http://www.dina.kvl.dk/~pmb/`.
3. R. M. Cohen. The defensive java virtual machine specification version 0.5. Technical report, Computational Logic Inc, Austin, Texas, May 1997. `http://www.cli.com/`.
4. J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, Reading, Massachusetts, 1996.
5. P. H. Hartel. LATOS – a lightweight animation tool for operational semantics. Technical report DSSE-TR-97-1, Dept. of Electr. and Comp. Sci, Univ. of Southampton, England, Oct 1997. `http://www.ecs.soton.ac.uk/~phh/latos.html`.
6. M. Levy. *Java Secure processor language specification version 0.99*. Integrity Arts Inc., San Mateo, California, May 1997.
7. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison Wesley, Reading, Massachusetts, 1996.
8. G. McGraw and E. W. Felten. *Java security: Hostile applets, holes and antidotes*. John Wiley & Sons, Chichester, England, 1997.
9. P. Peyret. Application-enabling card systems with plug-and-play applets. In *Smart Card 1996 convention proceedings – Technology and markets conference*, pages 51–72. Quality marketing services Ltd, Peterborough, UK, Feb 1996.
10. R. Stata and M. Abadi. A type system for Java bytecode subroutines. In *25th Principles of programming languages (POPL)*, San Diego, California, Jan 1998. ACM, New York.
11. D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In J.-P. Jouannaud, editor, *2nd Functional programming languages and computer architecture, LNCS 201*, pages 1–16, Nancy, France, Sep 1985. Springer-Verlag, Berlin.
12. J. L. Zoreda and J. M. Otón. *Smart Cards*. Artech House Inc, Norwood, Massachusetts, 1994.