Bytecode Verification and its Applications

April 14, 2006

Contents

1 Introduction

2 Java Bytecode Language and its Operational semantics

The purpose of this section is to introduce the fundamental concepts of the present thesis. In particular, we present a language of bytecode instructions and a language of expressions that the bytecode manipulates. The bytecode language supports the basic features of the Java Virtual Machine [12](JVM for short), namely method invokations, object manipulation and creation as well as exceptions. Finally, we give a big step operational semantics of the bytecode language in terms of program state transitions.

In the following, subsection 2.1 is an overview of existing formalisations of the JVM semantics, subsection 2.2 gives some particular notations that will be used from now on along the thesis, subsection 2.3 introduces the structures classes, fields and methods used in the virtual machine, subsection 2.4 gives the type system which is supported by the bytecode language, subsection 2.5 describes the expressions that our bytecode language manipulates, subsection 2.6 gives the modelisation of the memory heap, subsection 2.7 introduces the notion of state configuration, subsection 2.8 gives the operational semantics of our language.

2.1 Related Work

A considerable effort has been done on the formalization of the semantics of the JVM. Most of the existing formalizations cover a representative subset of the language. Among them is the work [8] by N.Freund and J.Mitchell and [14] by Qian, which give a formalization in terms of a small step operational semantics of a large subset of the Java bytecode language including method calls, object creation and manipulation, exception throwing and handling as well subroutines, which for the formal specification of the language and the bytecode verifier.

Based on the work of Qian, in [13] C.Pusch gives a formalization of the JVM and the Java Bytecode Verifier in Isabelle/HOL and proves in it the soundness of the specification of the verifier. In [10], Klein and Nipkow give a formal small step and big step operational semantics of a Java-like language called Jinja, an operational semantics of a Jinja VM and its type system and a bytecode verifier as well a compiler from Jinja to the language of the JVM. They prove the equivalence between the small and big step semantics of Jinja, the type safety for the Jinja VM, the correctness of the bytecode verifier w.r.t. the type system and finally that the compiler preseves semantics and well-typedness.

The small size and complexity of the JavaCard platform simplifies the formalization of the system and thus, has attracted particularly the scientific interest. CertiCartes [3, 2] is an in-depth formalization of JavaCard. It has a formal executable specification written in Coq of a defensive and an offensive JCVM and an abstract JCVM together with the specification of the Java Bytecode Verifier.

2.2 Notation

In the following, if we have a function f with domain type A and range type B we note it with $f : A \rightarrow B$. If the function receives n arguments of type $A1 \ \ldots \ A2$ respectively and maps them to elements of type B we note the function signature with $f : A1 * ... * A2 \rightarrow B$. Function updates of function f with n arguments is denoted with $f[\oplus x1 \ldots xn \rightarrow y]$ and the definition of such function is :

$$f[\oplus x1 \ldots xn \rightarrow y](z1 \ldots zn) = \begin{cases} y & if\ x1 = z1 \wedge ... \wedge xn = zn \\ f(z1 \ldots zn) & else \end{cases}$$

The function inList? takes as arguments any list and an object and returns *true* if the object is in the list and *false* otherwise:

$$\text{inList?} : list\ A * A \rightarrow\ bool$$

For any type A, the function ++ takes as argument any list $l : list\ A$ and an object $o : A$ and returns a list $l1$ such that $l1.head = o \wedge l1.tail = l$:

$$++ : list\ \text{RefType} * \text{RefType} \rightarrow list\ \text{RefType}$$

2.3 Classes, Fields and Methods

Java programs are a set of classes. As the JVM says *A class declaration specifies a new reference type and provides its implementation. ... The body of a class declares members (fields and methods), static initializers, and constructors.* In our formalisation, the set of classes is denoted with *Class*, the set of fields with *Field*, the set of methods *Method*. We define a domain for class names *ClassName*, for field names *FieldName* and for method names *MethodName* respectively.

An object of type *Class* is a tuple with the following components: list of field objects (*fields*), which are declared in this class, list of the methods declared in the class (*methods*), the name of the class (*className*) and the super class of the class (*superClass*) :

$$\forall C : Class,$$
$$C = \left\{ \begin{array}{ll} fields & : listField \\ methods & : listMethod \\ className & : ClassName \\ superClass & : Class \end{array} \right\}$$

A field object is a tuple that contains the unique field id and a field type, as well as where as the class where it is declared :

$$\forall f : Field,$$
$$f = \left\{ \begin{array}{ll} Name & : FieldName; \\ Type & : JType; \\ declaredIn & : JClass \end{array} \right\}$$

We introduce a special field which stands for the number of components of any reference pointing to an array object and which does not belong to any class

(the name of the object and its field *Name* have the same name):

$$\text{arrLength} = \left\{ \begin{array}{ll} \mathit{Name} & = \text{arrLength}; \\ \mathit{Type} & = \mathit{int}; \\ \mathit{declaredIn} & = \bot \end{array} \right\}$$

A method has a unique method id (*Name*), a return type (*retType*), a list containing the formal parameter names and their types(*args*), the number of its formal parameters (*nArgs*), list of bytecode instructions representing its body (*body*) and the entry point instruction of the method (*entryPnt*) (the instruction at which every execution of the method starts), the exception handler table (*excHndls*)

$$\forall m : \mathit{Method}, \\ m = \left\{ \begin{array}{ll} \mathit{Name} & : \mathit{MethodName} \\ \mathit{retType} & : \mathit{JType} \\ \mathit{args} & : (\mathit{name} * \mathit{JType})[] \\ \mathit{nArgs} & : \mathit{nat} \\ \mathit{body} & : \text{I}[] \\ \mathit{entryPnt} & : \text{I} \\ \mathit{excHndls} & : \mathit{ExcHandler}[] \end{array} \right\}$$

We assume that for every method $m$the entrypoint is the first instruction in the array of instructions of which its body consists, i.e. $m.\mathit{entryPnt} = m.\mathit{body}[0]$.

An object of type *ExcHandler* contains information about the region in the method body that it protects, i.e. the start position (*startPc*) of the region and the end position (*endPc*), about the exception it protects from (*exc*), as well as what position in the method body the exception handler starts (*handlerPc*) at.

$$\forall \mathit{excH} : \mathit{ExcHandler}, \\ \mathit{excH} = \left\{ \begin{array}{ll} \mathit{startPc} & : \mathit{nat} \\ \mathit{endPc} & : \mathit{nat} \\ \mathit{handlerPc} & : (\mathit{name} * \mathit{type})[] \\ \mathit{exc} & : \mathit{Class}_{\mathit{exc}} \end{array} \right\}$$

2.4 Program Types

The types supported by our language are a simplified version of the types supported by the JVM. Thus, we have a unique simple type : the integer data type int. The reference type (RefType) stands for the simple reference types (RefCl) and array reference types(RefArr) . The language does not support interface types, still it can be extended with minor difficuties to interface types.

$$\begin{array}{ll} \mathit{JType} & ::= \text{int} \mid \text{RefType} \\ \text{RefType} & ::= \text{RefCl} \mid \text{RefArr} \\ \text{RefCl} & ::= \mathit{Class} \\ \text{RefArr} & ::= \mathit{JType}[] \end{array}$$

Every type has an associated default value which can be accessed via the function *default*. The function is defined as follows

$$\mathit{default}\ \mathit{Val}(\mathtt{T}) = \left\{ \begin{array}{ll} \mathtt{null} & \mathtt{T} \in \text{RefType} \\ 0 & \mathtt{T} = \text{int} \end{array} \right.$$

We define also a subtyping relation as follows:

$$\frac{}{\textsf{subtype } (C,C)} \qquad \frac{C2{=}C1.superClass}{\textsf{subtype } (C1,C2)}$$

$$\frac{C3{=}C1.superClass \ \textsf{subtype } (C3,C2)}{\textsf{subtype } (C1,C2)} \qquad \frac{}{\textsf{subtype } (C1,Object)}$$

$$\frac{}{\textsf{subtype } (C[],Object)} \qquad \frac{\textsf{subtype } (C1,C2)}{\textsf{subtype } (C1[],C2[])}$$

2.5 Expressions

We are now going to look at the expressions that our bytecode language manipulates. Similarly to the Java Bytecode language, the values of our language are of primitive data type[1] or references where references can be either references to class instance objects or to array objects. The special reference `null` does not point to any object.

$$\begin{array}{rl} \text{Values} & := i, i \in \text{ intliteral } \mid \text{ RefVal} \\ \text{RefVal} & := Ref : \text{RefType} \mid \texttt{null} \end{array}$$

We introduce several notations which will be used in the following. If we want to say explicitely that a reference points to a class instance object of type C we denote this with $\texttt{ref}_C$ and if a reference points to a location of an array object whose elements are of type `T` we denote this with $\texttt{refArr}_{\texttt{T}[\,]}$. Fields are modeled as functions from references to values.

Our language supports field access expressions $f(\mathcal{E})$. When assigning value v to the object field f of the object pointed by the reference r, the corresponding field function is updated resulting in a new field function $f[\oplus r \longrightarrow v] : \text{RefType} \longrightarrow \text{Values}$. Next, our language supports arrays and the access to the i^{th} element in the array arr is denoted with $\text{arrAccess}(\text{arr}, \text{i})$. Similarly, we use update functions for array access expressions $\text{arrAccess}(\text{arr}, \text{i})[\oplus(\text{ref}, \text{ind}) \rightarrow \text{val}]$.

The list of registers is denoted with reg . The i^{th} element of reg is denoted by reg_i. Note that reg_0 will stand for the current object, which corresponds to `this` in the Java language. The language deals also with arithmetic expressions that evaluate to an integer value and have the form $\mathcal{E}_1$ `op` $\mathcal{E}_2$, where $\mathcal{E}_1$ and $\mathcal{E}_2$ are also arithmetic expressions and *op* is an arithmetic operation symbol ranging in $+, -, div, rem, xor, xand$.

As the Java virtual machine is stack based we will also need expressions that model respectively the operand stack and the stack counter. Thus, the expressions cntr and stack stand respectively for the stack counter and the stack. With st(i), we denote the expression at the i-th position on the stack is.

The following definition gives the formal grammar for the expressions of the bytecode language.

Expression 1 (Formal Grammar of Expressions)

[1] we support only the integer primitive data type, which is sufficient for the purposes of this document

$$\texttt{op} \quad ::= + \mid - \mid div \mid rem \mid xor \mid xand$$

$$\begin{array}{ll} \mathcal{E} & ::= \\ & \text{Values} \\ & \mid f(\mathcal{E}) \\ & \mid f[\oplus\mathcal{E} \rightarrow \mathcal{E}](\mathcal{E}) \\ & \mid \text{arrAccess}(\mathcal{E}, \mathcal{E}) \\ & \mid \text{arrAccess}[\oplus(\mathcal{E}, \mathcal{E}) \rightarrow \mathcal{E}](\mathcal{E}, \mathcal{E}) \\ & \mid \text{reg}_\text{i} \\ & \mid \mathcal{E} \ \texttt{op} \ \mathcal{E} \\ & \mid \text{cntr} \\ & \mid \text{st}(\mathcal{E}) \end{array}$$

2.5.1 Substitution

Expression substitution is defined inductively in a standard way over the expression structure. Still, we allow also substitution over objects that are not from our language, i.e. we apply substitution over field functions which result in an update version of this field function. This is done by establishing the substitution rule for field access as follows:

$$\texttt{f(o)}[\mathcal{E}_1 \leftarrow \mathcal{E}_2] = \texttt{f}[\mathcal{E}_1 \leftarrow \mathcal{E}_2](\texttt{o}[\mathcal{E}_1 \leftarrow \mathcal{E}_2])$$

In the next, we define a substitution over field function objects:

$$\texttt{f}[\mathcal{E}_1 \leftarrow \mathcal{E}_2] = \begin{cases} \texttt{f} & if \ \mathcal{E}_1 \neq \texttt{f} \\ \mathcal{E}_2 & if \ \mathcal{E}_1 = \texttt{f} \wedge \\ & \mathcal{E}_2 = \texttt{f}[\oplus \ \texttt{r} \ \longrightarrow \ \texttt{v} \ ] \end{cases}$$

$$\texttt{f}[\oplus \ \texttt{r'} \ \longrightarrow \ \texttt{v'} \ ][\mathcal{E}_1 \leftarrow \mathcal{E}_2] = \begin{cases} \texttt{f}[\oplus \ \texttt{r} \ [\mathcal{E}_1 \leftarrow \mathcal{E}_2] \longrightarrow \ \texttt{v} \ [\mathcal{E}_1 \leftarrow \mathcal{E}_2]] & if \ \mathcal{E}_1 \neq \texttt{f} \\ \texttt{f} \ \begin{array}{l}[\oplus \ \texttt{r'}[\mathcal{E}_1 \leftarrow \mathcal{E}_2] \longrightarrow \ \texttt{v'}[\mathcal{E}_1 \leftarrow \mathcal{E}_2]] \\ [\oplus \ \texttt{r} \ \longrightarrow \ \texttt{v} \ ]\end{array} & \begin{array}{l} if \ \mathcal{E}_1 = \texttt{f} \wedge \\ \mathcal{E}_2 = \texttt{f}[\oplus \ \texttt{r} \ \longrightarrow \ \texttt{v} \ ] \end{array} \end{cases}$$

For example, consider the following substitution

$$\texttt{f(o)}[\texttt{f} \leftarrow \texttt{f}[\oplus\texttt{e} \longrightarrow \texttt{v}]]$$

This results in the new expression :

$$\texttt{f}[\oplus\texttt{e} \longrightarrow \texttt{v}](\texttt{o}[\texttt{f} \leftarrow \texttt{f}[\oplus\texttt{e} \longrightarrow \texttt{v}]])$$

2.6 Modeling the Object Heap

An important issue for the modelization of an object oriented programming language and its operational semantics is the garbage collected memory heap.

As the Java Virtual Machine (JVM) specification states, the heap is the runtime data area from which memory for all class instances and arrays is allocated. Whenever a new instance is allocated, the JVM returns a reference value that points to the newly created object. We introduce a record type HeapType which models the memory heap. It consists of three components:

- a component named hFld which is a partial function that maps field structures (of type *Field* introduced in subsection 2.3) into functions from references (RefType) into values (Values).
- a component hArr which maps the components of arrays into their values
- a component hLoc which stands for the list of references that the heap has

More formally, the data type HeapType has the following structure:

$$\forall \mathrm{H} : \mathsf{HeapType},$$
$$\mathrm{H} = \left\{ \begin{array}{ll} \mathsf{hFld} & : \mathit{Field} \rightarrow (\mathrm{RefType} \rightarrow \mathrm{Values}) \\ \mathsf{hArr} & : \mathrm{RefArr} * \mathit{nat} \rightarrow \mathrm{Values} \\ \mathsf{hLoc} & : \mathit{list}\ \mathrm{RefType} \end{array} \right\}$$

A heap object H must assure that the value of the components H.hFld and H.hArr are functions which are defined only for references from the proper type and which are in the list of references of the heap H.hLoc:

$$\begin{array}{ll} \forall f : \mathit{Field}, \forall\ \mathtt{ref}_C : \mathrm{RefType}, & \mathit{InDom}\ (\mathrm{H}.\mathsf{hFld}(f),\ \mathtt{ref}_C) \Rightarrow \\ & \mathrm{inList?}(\ \mathtt{ref}_C, \mathrm{getLoc(H)}\ ) \wedge \\ & f.\mathit{Type} = C \\ \wedge & \\ \forall\ \mathtt{ref}_{\mathtt{T}[]} : \mathrm{RefArr}, & \mathit{InDom}\ (\mathrm{H}.\mathsf{hArr}, (\ \mathtt{ref}_{\mathtt{T}[]}, i)) \Rightarrow \\ & \mathrm{inList?}(\mathrm{H}.\mathsf{hLoc},\ \mathtt{ref}_{\mathtt{T}[]}) \wedge \\ & 0 \leq i < \mathrm{H}.\mathsf{hFld}(\mathrm{arrLength})(\ \mathtt{ref}_{\mathtt{T}[]}) \end{array}$$

We define an operation addNewLoc which adds a new reference to the list of references in a heap. The only change that the operation will cause to the heap H is to add a new reference ref to the list of references of the heap H.hLoc:

$$\mathrm{addNewLoc}\ : \mathrm{H} * \mathrm{RefType} \rightarrow \mathrm{H}$$

Formally, the operation is defined as follows:

$$\begin{array}{ll} \mathrm{addNewLoc(H},\ \mathtt{ref}\ ) = \mathrm{H}' \iff^{def} & \\ & \\ \mathrm{H}.\mathsf{hLoc} = l \wedge & \\ \mathrm{inList?}(l,\ \mathtt{ref}\ ) = \mathit{false} \wedge & \\ \mathrm{H}'.\mathsf{hLoc} = \mathtt{ref} +\!\!+ l \wedge & \\ \forall f : \mathit{Field},\ \forall\ \mathtt{ref}\ : \mathrm{RefType}, & \mathrm{inList?}(l,\ \mathtt{ref}\ ) = \mathit{true} \wedge \\ & f.\mathit{declaredIn} = \mathit{typeof}(\ \mathtt{ref}\ ) \Rightarrow \\ & (\mathrm{H}.\mathsf{hFld}(f))(\ \mathtt{ref}\ ) = (\mathrm{H}'.\mathsf{hFld}(f))(\ \mathtt{ref}\ ) \\ \wedge & \\ \forall\ \mathtt{ref}\ : \mathrm{RefArr}, & \mathrm{inList?}(l,\ \mathtt{ref}\ ) = \mathit{true} \wedge \\ & \forall i, 0 \leq i < \mathrm{H}.\mathsf{hFld}(\mathrm{arrLength})(\ \mathtt{ref}\ ) \Rightarrow \\ & \mathrm{H}.\mathsf{hArr}(\ \mathtt{ref}\ , i) = \mathrm{H}'.\mathsf{hArr}(\ \mathtt{ref}\ , i) \end{array}$$

Whenever the field f for the object pointed by reference $\texttt{ref}$ is updated with the value *val*, the component H.hFld is updated:

$$\text{H.hFld} := \text{H.hFld}[\oplus f \rightarrow \text{H.hFld}(f)[\oplus \ \texttt{ref} \ \rightarrow val]]$$

In the following for the sake of clarity, we will use another lighter notation for a field update which do not imply any ambiguities:

$$\text{H}[\oplus f \rightarrow f[\oplus \ \texttt{ref} \ \rightarrow val]]$$

If in the heap H the i^{th} component in the array referenced by $\texttt{ref}_{\texttt{T}[]}$ is updated with the new value *val*, this results in assigning a new value of the component H.hArr:

$$\text{H.hArr} := \text{H.hArr}[\oplus(\ \texttt{ref}_{\texttt{T}[]}, i) \rightarrow val]$$

In the following for the sake of clarity, we will use another lighter notation for an update of an array component which do not imply any ambiguities:

$$\text{H}[\oplus(\ \texttt{ref}_{\texttt{T}[]}, i) \rightarrow val]$$

If a new object of class C is created in the memory, a fresh reference $\texttt{ref}_C$ which points to the newly created object is added in the heap H and all the values of the field functions that correspond to the fields in class C are updated for the new reference with the default values for their corresponding types. We denote the new heap with newRef(H, C). The formalization of the resulting heap and the new reference is the following:

$$\text{newRef(H, } C) = (\text{H}', \ \texttt{ref}_C) \iff^{def}$$

$$\left.\begin{array}{l} \text{addNewLoc(H, } \texttt{ref}_C) \ = \text{H}' \wedge \\ \left\{ \begin{array}{ll} \forall f : \mathit{Field}, & f.\mathit{declaredIn} = C \Rightarrow \\ & \text{H}'.\text{hFld} := \text{H}'.\text{hFld}[\oplus f \rightarrow \text{H}'.\text{hFld}(f)[\oplus \ \texttt{ref} \ \rightarrow \mathit{default}^{\ Val}(f.\mathit{Type})]] \end{array} \right. \end{array}\right\}$$

Identically, when allocating a new object of array type whose elements are of type Tand length is l, we obtain a new heap object newArrRef(H, T[ ], l) which is defined similarly to the previous case:

$$\text{newArrRef(H, T[ ]}, l) \ = (\text{H}', \ \texttt{ref}_{\texttt{T}[\ ]}) \iff^{def}$$

$$\begin{array}{l} \text{addNewLoc(H, } \texttt{ref}_{\texttt{T}[\ ]}) \ = \text{H}' \wedge \\ \text{H}'.\text{hFld} := \text{H}'.\text{hFld}[\oplus \text{arrLength} \rightarrow \text{H}'.\text{hFld}(\text{arrLength})[\oplus \ \texttt{ref}_{\texttt{T}[\ ]} \rightarrow l]] \wedge \\ \forall i, 0 \geq i < l \Rightarrow \text{H}'.\text{hArr} := \text{H}'.\text{hArr}[\oplus(\ \texttt{ref}_{\texttt{T}[]}, i) \rightarrow \mathit{default}^{\ Val}(\texttt{T})] \end{array}$$

2.7 State configuration

In this section, we introduce the notion of state configuration. A state configuration Kmodels the program state in particular execution program point by specifying what is the state of the memory heap, the stack and the stack

counter, the values of the local variables of the currently executed method and what is the instruction which is executed next.

We define two kinds of state configurations:

$$K = K^{interm} \cup K^{final}$$

The set K^{interm} consists of method intermediate state configurations, which stand for a *not stuck state* in which the execution of the current method is not finished i.e. there is still another instruction of the method body to be executed. The configuration $< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ > \in K^{interm}$ has the following elements:

- the function His a mapping from field function names to the corresponding field function and from references to objects in the memory heap.
- cntr is a variable that contains a natural number which stands for the number of elements in the operand stack.
- st stands for the operand stack and which for any integer `ind` smaller than the operand stack counter cntr retruns the value st(`ind`) stored in the operand stack at `ind` positions of the bottom of the stack. st is not defined for natural values greater than the counter. A newly created stack is denoted with [ ]
- reg is the array of local variables of a method and for an index `i` returns the value $\mathrm{reg_i}$ which is stored at that index of the array
- pc stands for the program counter and contains the index of the instruction to be executed in the current state

The elements of the set K^{final} are the final states, states in which the current method execution is terminated and consists of normal termination states (K^{norm}) and exceptional termination states (K^{exc}):

$$K^{final} = K^{norm} \cup K^{exc}$$

A method may terminate either normally (by reaching a return instruction) or exceptionally (by throwing an exception).

- $< \mathrm{H}, \mathrm{Res} >^{norm} \in K^{norm}$ which describes a *normal final state*, i.e. the method execution terminates normally. The normal termination configuration has the following components :
  - H reflects what is the heap state after the method terminated
  - Res stands for the return value of the method
- $< \mathrm{H}, \mathrm{Exc} >^{exc} \in K^{exc}$ which stands for an *exceptional final state* of a method, i.e. the method terminates by throwing an exception. The exceptional configuration has the following components:
  - the heap H
  - Exc is a reference to the uncaught exception that caused the method termination

We will use the notation $< \mathrm{H}, \mathit{Final} >^{final}$ for any configuration which belongs to the set K^{final}. In the next subsection, we define in terms of state configuration transition relation the operational semantics of our bytecode programming language.

2.8 Bytecode Language and its Operational Semantics

The bytecode language that we introduce here corresponds to a representative subset of the Java bytecode language. In particular, it supports object manipulation and creation, method invokation, as well as exception throwing and handling. In fig. 1, we give the list of instructions that constitute our bytecode language.

$$
\begin{array}{rl}
\mathrm{I} ::= & \mathrm{if\_cond} \\
| & \mathrm{goto} \\
| & \mathrm{return} \\
| & \mathrm{arith\_op} \\
| & \mathrm{load} \\
| & \mathrm{store} \\
| & \mathrm{push} \\
| & \mathrm{pop} \\
| & \mathrm{dup} \\
| & \mathrm{iinc} \\
| & \mathrm{new} \\
| & \mathrm{newarray} \\
| & \mathrm{putfield} \\
| & \mathrm{getfield} \\
| & \mathrm{type\_astore} \\
| & \mathrm{type\_aload} \\
| & \mathrm{arraylength} \\
| & \mathrm{instanceof} \\
| & \mathrm{checkcast} \\
| & \mathrm{athrow} \\
| & \mathrm{invoke}
\end{array}
$$

Figure 1: Bytecode Language instructions

Note that the instruction arith_op stands for any arithmetic instruction in the list add , sub , mult , and , or , xor , ishr , ishl , div , rem).

JVM is stack based and a new method is called a new method frame is pushed on the frame stack and the execution continues on this new frame. A method frame contains the method operand stack, the array of registers and the constant pool of the class the method belongs to. When a method terminates its execution normally, the result, if any, is popped from the method operand stack, the method frame is popped from the frame stack and the method result (if any) is pushed on the operand stack of its caller. If the method terminates with an exception, it does not return any result and the exception object is propagated back to its callers.

Our formalization considers only single method execution abstracting from the method frame stack as it is sufficient for the proof of correctness of the definition of the weakest precondition calculus. This abstraction is possible as the proposed verification procedure is modular - when verifying a method we assume that all the other methods are correct and we are concerned only with establishing the correctness of the current method.

We define the operational semantics of a single Java instruction in terms of relation between the instruction and the state configurations before and after its execution.

Definition 2.1 (State Transition) *If an instruction* I *in the body of method* m *starts execution in a state with configuration* $< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ >$ *and terminates execution in state with configuration* $< \mathrm{H}', \mathrm{cntr}', \mathrm{st}\ ', \mathrm{reg}\ ', \mathrm{pc}\ ' >$ *we denote this by*

$$m \vdash \ \mathrm{I} :< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ > \longrightarrow < \mathrm{H}', \mathrm{cntr}', \mathrm{st}\ ', \mathrm{reg}\ ', \mathrm{pc}\ ' >$$

We also define how the execution of a list of instructions change the state configuration in which their execution starts.

Definition 2.2 (Transitive closure of a state transition relation) *If the body of the method* m $m.body$*starts execution in a state with configuration* $< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ >$ *and there is an execution path from* m *.entryPnt to an instruction* m *.body[k] which is either a* return *instruction or an instruction which terminates execution with an uncaught exception and the configuration after its execution is* $< \mathrm{H}', \mathit{Final} >^{final}$ *we denote it with*

$$m \vdash m.body :< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ > \longrightarrow^* < \mathrm{H}', \mathit{Final} >^{final}$$

We first give the operational semantics of a method execution. The execution of method $m$is the execution of its body upto reaching a final state configuration:

$$\frac{m.body :< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ > \longrightarrow^* < \mathrm{H}', \mathit{Final} >^{final}}{m :< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ > \longrightarrow < \mathrm{H}', \mathit{Final} >^{final}}$$

The function *getStateOnExc* deals with bytecode instructions that may throw exceptions. The function returns the state configuration after the current instruction during the execution of m throws an exception of type E. If the method m has an exception handler which can handle exceptions of type E thrown at the index of the current instruction, the execution is not stuck and thus, the state configuration is an intermediate state configuration. If the method m does not have an exception handler for dealing with exceptions of type E at the current index, the execution of m terminates exceptionally and the current instruction causes the method exceptional termination:

$$\mathit{getStateOnExc}\ : K^{interm} * \mathit{ExcType} * \mathit{ExcHandler}[] \rightarrow K^{interm} \cup K^{exc}$$

$$\mathit{getStateOnExc}\ (< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ >, E, \mathrm{pc}\ , \mathit{excH}[]) =$$
$$\begin{cases} < \mathrm{H}', 0, \mathrm{st}\ [\oplus 0 \rightarrow \ \mathtt{ref}_E], \mathrm{reg}\ , \mathit{handlerPc} > & \text{if } \mathit{findExcHandler}(\mathrm{E},\ \mathrm{pc}\ ,\ \mathit{excH}[]) = \mathit{handlerPc} \\ < \mathrm{H}',\ \mathtt{ref}_E >^{exc} & \text{if } \mathit{findExcHandler}(\mathrm{E},\ \mathrm{pc}\ ,\ \mathit{excH}[]) = \bot \end{cases}$$

where
$(\mathrm{H}',\ \mathtt{ref}_E) = \mathrm{newRef}(\mathrm{H}, E)$

If an exception E is thrown by instruction at position i while executing the method m, the exception handler table $m.excHndls$ will be searched for the first exception handler that can handle the exception. The search is done by the function *findExcHandler* . If there is found such a handler the function returns the index of the instruction at which the exception handler starts, otherwise it returns $\perp$:

$$findExcHandler \ : ExcType * nat * ExcHandler[] \rightarrow nat$$

$$findExcHandler(E, \text{pc} \ , excH[]) = \begin{cases} handlerPc & \begin{cases} \exists k, min(k) : excH[k] = (startPc, endPc, handlerPc, E') \wedge \\ startPc \leq \text{pc} \ < endPc \wedge \\ \text{subtype} \ (E, E') \end{cases} \\ \perp & \text{else} \end{cases}$$

Next, we define the operational semantics of every instruction. The operational semantics of an instruction states how the execution of an instruction affects the program state configuration in terms of state configuration transitions defined in the previous subsection 2.7. Note that we do not model the method frame stack of the JVM which is not needed for our purposes.

- Control transfer instructions

1. Conditional jumps : if_cond

$$\frac{\texttt{cond}(\text{st(cntr)},\text{st(cntr}-1))}{\text{m} \vdash \text{if\_cond} \ \ _{n:}<\text{H},\text{cntr},\text{st} \ ,_{\text{reg}} \ ,\text{pc} \ > \longrightarrow <\text{H},\text{cntr}-2,\text{st} \ ,_{\text{reg}} \ ,\text{pc} \ +n>}$$

$$\frac{_{not(} \ \ \texttt{cond}(\text{st(cntr)},\text{st(cntr}-1)))}{\text{m} \vdash \text{if\_cond} \ \ _{n:}<\text{H},\text{cntr},\text{st} \ ,_{\text{reg}} \ ,\text{pc} \ > \longrightarrow <\text{H},\text{cntr}-2,\text{st} \ ,_{\text{reg}} \ ,\text{pc} \ +1>}$$

   The condition `cond` is applied to the stack top and zero. If the condition is true then the control is transfered to the instruction at index `n`, otherwise the control continues at the instruction following the current instruction. The top two elements st(cntr) and st(cntr - 1) of the stack top are popped from the operand stack.

2. Unconditional jumps: goto

$$\frac{}{\text{m} \ \vdash \ \text{goto} \ \ n :< \text{H}, \text{cntr}, \text{st} \ , \text{reg} \ , \text{pc} \ > \longrightarrow < \text{H}, \text{cntr}, \text{st} \ , \text{reg} \ , \text{pc} \ + n >}$$

   Transfers control to the instruction at position n.

3. return

$$\frac{}{\text{m} \ \vdash \ \text{return} \ :< \text{H}, \text{cntr}, \text{st} \ , \text{reg} \ , \text{pc} \ > \longrightarrow < \text{H}, \text{st(cntr)} >^{norm}}$$

   The instruction causes the normal termination of the execution of the current method m. The instruction does not affect changes on the heap Hand the return result is contained in the stack top element st(cntr)

- Arithmetic operations

$$\frac{\begin{array}{l}\text{cntr}' = \text{cntr} - 1\\ \text{st}\,' = \text{st}\,[\oplus\text{cntr} - 1 \rightarrow \text{st}(\text{cntr} - 1)\ \text{op}\ \text{st}(\text{cntr})]\\ \text{pc}\,' = \text{pc}\ + 1\end{array}}{\text{m}\ \vdash\ \texttt{op}\ :< \text{H}, \text{cntr}, \text{st}\ , \text{reg}\ , \text{pc}\ > \longrightarrow < \text{H}, \text{cntr}', \text{st}\,', \text{reg}\ , \text{pc}\,' >}$$

Pops the values which are on the stack top st(cntr) and st(cntr - 1) at the position below and applies the arithmetic operation `op` on them. The stack counter is decremented and the resulting value on the stack top st(cntr - 1) op st(cntr) is pushed on the stack top st(cntr - 1).

case for arithmetic instructions that throw exception

- Load Store instructions

1. load

$$\frac{\begin{array}{l}\text{cntr}' = \text{cntr} + 1\\ \text{st}\,' = \text{st}\,[\oplus\text{cntr} + 1 \rightarrow \text{reg}_\text{i}]\\ \text{pc}\,' = \text{pc}\ + 1\end{array}}{\text{m}\ \vdash\ \text{load}\ \ \text{i} :< \text{H}, \text{cntr}, \text{st}\ , \text{reg}\ , \text{pc}\ > \longrightarrow < \text{H}, \text{cntr}', \text{st}\,', \text{reg}\ , \text{pc}\,' >}$$

The instruction loads increments the stack counter cntr and pushes the content of the local variable reg_i on the stack top st(cntr + 1)

2. store

$$\frac{\begin{array}{l}\text{cntr}' = \text{cntr} - 1\\ \text{reg}\,' = \text{reg}\ [\oplus\ \text{i} \rightarrow \text{st}(\text{cntr})]\\ \text{pc}\,' = \text{pc}\ + 1\end{array}}{\text{m}\ \vdash\ \text{store}\ \ \text{i} :< \text{H}, \text{cntr}, \text{st}\ , \text{reg}\ , \text{pc}\ > \longrightarrow < \text{H}, \text{cntr}', \text{st}\ , \text{reg}\,', \text{pc}\,' >}$$

Pops the stack top element st(cntr) and stores it into local variable reg_i and decrements the stack counter cntr

3. iinc

$$\frac{\begin{array}{l}\text{reg}\,' = \text{reg}\ [\oplus\ \text{i} \rightarrow \text{reg}_\text{i} + 1]\\ \text{pc}\,' = \text{pc}\ + 1\end{array}}{\text{m}\ \vdash\ \text{iinc}\ \ \ \text{i} :< \text{H}, \text{cntr}, \text{st}\ , \text{reg}\ , \text{pc}\ > \longrightarrow < \text{H}, \text{cntr}, \text{st}\ , \text{reg}\,', \text{pc}\,' >}$$

Increments the value of the local variable reg_i

4. push

$$\frac{\begin{array}{l}\text{cntr}' = \text{cntr} + 1\\ \text{st}\,' = \text{st}\,[\oplus\text{cntr} + 1 \rightarrow\ \text{i}]\\ \text{pc}\,' = \text{pc}\ + 1\end{array}}{\text{m}\ \vdash\ \text{push}\ \ \text{i} :< \text{H}, \text{cntr}, \text{st}\ , \text{reg}\ , \text{pc}\ > \longrightarrow < \text{H}, \text{cntr} + 1, \text{st}\,', \text{reg}\ , \text{pc}\,' >}$$

5. pop

$$\frac{}{\text{m}\ \vdash\ \text{pop} :< \text{H}, \text{cntr}, \text{st}\ , \text{reg}\ , \text{pc}\ > \longrightarrow < \text{H}, \text{cntr} + 1, \text{st}\ , \text{reg}\ , \text{pc}\ + 1 >}$$

- Object creation and manipulation

  1. new `Cl`

$$\frac{\begin{array}{l}(\mathrm{H}',\ \mathtt{ref}_C) = \mathrm{newRef}(\mathrm{H}, C)\\ \mathrm{cntr}' = \mathrm{cntr} + 1\\ \mathrm{st}\,' = \mathrm{st}\ [\oplus \mathrm{cntr} + 1 \rightarrow\ \mathtt{ref}_C]\\ \mathrm{pc}\,' = \mathrm{pc}\ + 1\end{array}}{\mathrm{m}\ \vdash\ \mathrm{new}\ C :< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ > \longrightarrow < \mathrm{H}', \mathrm{cntr}', \mathrm{st}\,', \mathrm{reg}\ , \mathrm{pc}\,' >}$$

  A new fresh location $\mathtt{ref}_C$ is added in the memory heap, the stack counter cntr is incremented. and $\mathtt{ref}_C$ is put on the stack top st(cntr + 1).

  2. putfield

$$\frac{\begin{array}{l}\mathrm{st}(\ \mathrm{cntr} - 1) \neq \mathtt{null}\ \Rightarrow\\ \begin{cases}\mathrm{H}' = \mathrm{H}[\oplus f \rightarrow \mathrm{H}(f)[\oplus \mathrm{st}(\ \mathrm{cntr}) \rightarrow \mathrm{st}(\ \mathrm{cntr} - 2)]]\\ \mathrm{cntr}' = \mathrm{cntr} - 2\\ \mathrm{pc}\,' = \mathrm{pc}\ + 1\end{cases}\end{array}}{m \vdash\ \mathrm{putfield}\ f :< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ > \longrightarrow < \mathrm{H}', \mathrm{cntr}', \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\,' >}$$

$$\frac{\begin{array}{l}\mathrm{st}(\ \mathrm{cntr} - 1) == \mathtt{null}\ \Rightarrow\\ \mathit{getStateOnExc}\ (< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ >,\ \mathtt{NullPntrExc}, m.\mathit{excHndls}) = k\end{array}}{m \vdash\ \mathrm{putfield}\ f :< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ > \longrightarrow k}$$

  The top value contained on the stack top st(cntr) and the reference contained in st(cntr - 1) are popped from the operand stack. If st(cntr - 1) is not `null` [2] , the value of its field `f` for the object is updated with the valuest(cntr) and the counter cntr is decremented. If the reference in st(cntr - 1) is `null` then a `NullPntrExc` is thrown

  3. getfield

$$\frac{\begin{array}{l}\mathrm{st}(\ \mathrm{cntr}) \neq \mathtt{null}\ \Rightarrow\\ \begin{cases}\mathrm{st}\,' = \mathrm{st}\ [\oplus \mathrm{cntr} \rightarrow f(\mathrm{st}(\ \mathrm{cntr}))]\\ \mathrm{pc}\,' = \mathrm{pc}\ + 1\end{cases}\end{array}}{m \vdash\ \mathrm{getfield}\ \mathtt{f}^c_f :< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ > \longrightarrow < \mathrm{H}, \mathrm{cntr}, \mathrm{st}\,', \mathrm{reg}\ , \mathrm{pc}\,' >}$$

$$\frac{\begin{array}{l}\mathrm{st}(\ \mathrm{cntr}) == \mathtt{null}\ \Rightarrow\\ \mathit{getStateOnExc}\ (< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ >,\ \mathtt{NullPntrExc}, m.\mathit{excHndls}) = k\end{array}}{m \vdash\ \mathrm{getfield}\ \mathtt{f}^c_f :< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ > \longrightarrow k}$$

  The top stack element st(cntr) is popped from the stack. If st(cntr) is not `null` the value of the field `f` in the object referenced by

[2] here we assume that the code has passed successfully the bytecode verification procedure and thus, for instance, st(cntr - 1) contains certainly a reference of type `C`

the reference contained in st(cntr), is fetched and pushed onto the operand stack st(cntr). If st(cntr) is `null` then a `NullPointerExc` is thrown, i.e. the stack counter is set to 0, a new object of type `NullPointerExc` is created in the memory heap store Hand a reference to it $\texttt{ref}_{NullPointerExc}$ is pushed onto the operand stack

4. newarray `T`

$$\frac{\begin{array}{l}\text{st(cntr)} \geq 0 \Rightarrow \\ \left\{\begin{array}{l}(\text{H}', \texttt{refArr}_{\texttt{type}}) = \text{newArrRef(H}, \texttt{type}, \text{st(cntr)}) \\ \text{cntr}' = \text{cntr} + 1 \\ \text{st}\,' = \text{st}\,[\oplus \text{cntr} + 1 \rightarrow \texttt{refArr}_{\texttt{type}}] \\ \text{pc}\,' = \text{pc}\, + 1\end{array}\right.\end{array}}{m \vdash \text{ newarray } \texttt{T} :< \text{H, cntr, st }, \text{reg }, \text{pc }\ > \longrightarrow < \text{H}', \text{cntr}', \text{st }', \text{reg }, \text{pc }' >}$$

$$\frac{\begin{array}{l}\text{st(cntr)} < 0 \Rightarrow \\ \textit{getStateOnExc } (< \text{H, cntr, st }, \text{reg }, \text{pc }\ >,\ \texttt{NegArrSizeExc}, m.\textit{excHndls}) = k\end{array}}{m \vdash \text{ newarray } \texttt{T} :< \text{H, cntr, st }, \text{reg }, \text{pc }\ > \longrightarrow k}$$

A new array whose components are of type `T` and whose length is the stack top value is allocated on the heap. The array elements are initialised to the default value of `T` and a reference to it is put on the stack top. In case the stack top is less than 0, then `NegArrSizeExc` is thrown

5. type_astore

$$\frac{\begin{array}{l}\text{st(cntr} - 2) \neq \texttt{null} \wedge \\ 0 \leq \text{st(cntr} - 1) < \text{arrLength(st(cntr} - 2)) \Rightarrow \\ \left\{\begin{array}{l}\text{H}' = \text{H}[\oplus(\text{st(cntr} - 2), \text{st(cntr} - 1)) \rightarrow \text{st(cntr)}] \\ \text{cntr}' = \text{cntr} - 3 \\ \text{pc}\,' = \text{pc}\, + 1\end{array}\right.\end{array}}{m \vdash \text{ type\_astore } :< \text{H, cntr, st }, \text{reg }, \text{pc }\ > \longrightarrow < \text{H}', \text{cntr}', \text{st }, \text{reg }, \text{pc }' >}$$

$$\frac{\begin{array}{l}\text{st(}\ \text{cntr} - 1) == \texttt{null}\ \Rightarrow \\ \textit{getStateOnExc } (< \text{H, cntr, st }, \text{reg }, \text{pc }\ >,\ \texttt{NullPntrExc}, m.\textit{excHndls}) = k\end{array}}{m \vdash \text{ type\_astore } :< \text{H, cntr, st }, \text{reg }, \text{pc }\ > \longrightarrow k}$$

$$\frac{\begin{array}{l}(\text{st(cntr} - 1) < 0 \vee \\ \text{st(cntr} - 1) \geq \text{arrLength(st(cntr} - 2))) \Rightarrow \\ \textit{getStateOnExc } (< \text{H, cntr, st }, \text{reg }, \text{pc }\ >,\ \texttt{ArrIndBndExc}, m.\textit{excHndls}) = k\end{array}}{m \vdash \text{ type\_astore } :< \text{H, cntr, st }, \text{reg }, \text{pc }\ > \longrightarrow k}$$

The three top stack elements st(cntr), st(cntr - 1) and st(cntr - 2) are popped from the operand stack. The type value contained in st(cntr) must be assignment compatible with the type of the elements of the array reference contained in st(cntr - 2), st(cntr - 1) must be of type int.

say what assignment compatible is

The value st(cntr) is stored in the component at index st(cntr - 1) of the array in st(cntr - 2). If st(cntr - 2) is `null` a `NullPntrExc`is

thrown. If st(cntr - 1) is not in the bounds of the array in st(cntr - 2) an `ArrIndBndExc` exception is thrown. If st(cntr) is not assignment compatible with the type of the components of the array, then `ArrStoreExc` is thrown

one more case of exceptional termination if it terminates on an exception

6. type_aload

$$\frac{\left.\begin{array}{l}\text{st(cntr} - 1) \neq \texttt{null} \wedge \\ \text{st(cntr)} \geq 0 \wedge \\ \text{st(cntr)} < \text{arrLength(st(cntr} - 1))\end{array}\right\} \Rightarrow \quad \left\{\begin{array}{l}\text{cntr}' = \text{cntr} - 1 \\ \text{st}\,' = \text{st}\,[\oplus\text{cntr} - 1 \rightarrow \text{arrAccess(st(cntr} - 1), \text{st(cntr))}] \\ \text{pc}\,' = \text{pc} + 1\end{array}\right.}{m \vdash \text{type\_aload} :< \text{H, cntr, st , reg , pc} > \longrightarrow < \text{H, cntr}', \text{st}\,', \text{reg , pc}\,' >}$$

$$\frac{\begin{array}{l}\text{st(cntr} - 1) == \texttt{null} \Rightarrow \\ \mathit{getStateOnExc}\ (< \text{H, cntr, st , reg , pc} >,\ \texttt{NullPntrExc}, m.\mathit{excHndls}) = k\end{array}}{m \vdash \text{type\_aload} :< \text{H, cntr, st , reg , pc} > \longrightarrow k}$$

$$\frac{\begin{array}{l}\text{st(cntr} - 1) \neq \texttt{null} \wedge \\ (\text{st(cntr)} < 0 \vee \\ \text{st(cntr)} \geq \text{arrLength(st(cntr} - 1))) \Rightarrow \\ \mathit{getStateOnExc}\ (< \text{H, cntr, st , reg , pc} >,\ \texttt{ArrIndBndExc}, m.\mathit{excHndls}) = k\end{array}}{m \vdash \text{type\_aload} :< \text{H, cntr, st , reg , pc} > \longrightarrow k}$$

Loads a value from an array. The top stack element st(cntr) and the element below it st(cntr -1) are popped from the operand stack. st(cntr) must be of type int. The value in st(cntr -1) must be of type RefCl whose components are of type type. The value in the component of the array `arrRef` at index `ind` is retrieved and pushed onto the operand stack. If st(cntr -1) contains the value `null` a `NullPntrExc`is thrown. If st(cntr) is not in the bounds of the array object referenced by st(cntr -1) a `ArrIndBndExc` is thrown

7. arraylength

$$\frac{\text{st(cntr)} \neq \texttt{null} \Rightarrow \quad \left\{\begin{array}{l}\text{H}' = \text{H} \\ \text{cntr}' = \text{cntr} \\ \text{st}\,' = \text{st}\,[\oplus\text{cntr} \rightarrow \text{arrLength(st(cntr))}] \\ \text{pc}\,' = \text{pc} + 1\end{array}\right.}{m \vdash \text{arraylength} :< \text{H, cntr, st , reg , pc} > \longrightarrow < \text{H}', \text{cntr}', \text{st}\,', \text{reg , pc}\,' >}$$

$$\frac{\begin{array}{l}\text{st(cntr)} == \texttt{null} \Rightarrow \\ \mathit{getStateOnExc}\ (< \text{H, cntr, st , reg , pc} >,\ \texttt{NullPntrExc}, m.\mathit{excHndls}) = k\end{array}}{m \vdash \text{arraylength} :< \text{H, cntr, st , reg , pc} > \longrightarrow k}$$

The stack top element is popped from the stack. It must be a reference that points to an array. If the stack top element st(cntr) is not `null` the length of the array arrLengthst(cntr) is fetched and pushed on the stack. If the stack top element st(cntr) is `null` then a `NullPntrExc`is thrown.

8. instanceof

$$\frac{\begin{array}{l}\mathsf{subtype}\ (\mathrm{st(cntr)}, C) \Rightarrow \\ \mathrm{st}\,' = \mathrm{st}\ [\oplus \mathrm{cntr} \rightarrow 1] \\ \mathrm{pc}\,' := \mathrm{pc}\ + 1\end{array}}{\text{instanceof } C :< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ > \longrightarrow < \mathrm{H}, \mathrm{cntr}, \mathrm{st}\,', \mathrm{reg}\ , \mathrm{pc}\,' >}$$

$$\frac{\begin{array}{l}\mathit{not}(\mathsf{subtype}\ (\mathrm{st(cntr)}, C) \vee \mathrm{st(cntr)} == \texttt{null}\ \Rightarrow \\ \mathrm{st}\,' = \mathrm{st}\ [\oplus \mathrm{cntr} \rightarrow 0] \\ \mathrm{pc}\,' := \mathrm{pc}\ + 1\end{array}}{m \vdash\ \text{instanceof } \texttt{C} :< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ > \longrightarrow < \mathrm{H}, \mathrm{cntr}, \mathrm{st}\,', \mathrm{reg}\ , \mathrm{pc}\,' >}$$

The stack top is popped from the stack. If it is of subtype C or is `null` , then the `1` is pushed on the stack, otherwise `0`.

9. checkcast

$$\frac{\begin{array}{l}\mathsf{subtype}\ (\mathrm{st(cntr)}, \texttt{C}) \vee \mathrm{st(cntr)} == \texttt{null}\ \Rightarrow \\ \mathrm{pc}\,' = \mathrm{pc}\ + 1\end{array}}{m \vdash\ \text{checkcast } C :< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ > \longrightarrow < \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\,' >}$$

$$\frac{\begin{array}{l}\mathit{not}(\mathsf{subtype}\ (\mathrm{st(cntr)}, C) \Rightarrow \\ \mathit{getStateOnExc}\ (< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ >, \texttt{CastExc}, m.\mathit{excHndls}) = k\end{array}}{m \vdash\ \text{checkcast } C :< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ > \longrightarrow k}$$

The stack top is popped from the stack. If it is not of subtype `C` an exception of type `CastExc`is thrown.

- Throw exception instruction. athrow

$$\frac{\begin{array}{l}\mathrm{st(cntr)} \neq \texttt{null}\ \Rightarrow \\ \mathit{getStateOnExc}\ (< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ >, \mathit{typeof}(\mathrm{st(cntr)}), m.\mathit{excHndls}) = k\end{array}}{m \vdash\ \text{athrow} :< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ > \longrightarrow k}$$

$$\frac{\begin{array}{l}\mathrm{st(cntr)} == \texttt{null}\ \Rightarrow \\ \mathit{getStateOnExc}\ (< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ >, \texttt{NullPntrExc}, m.\mathit{excHndls}) = k\end{array}}{m \vdash\ \text{athrow} :< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ > \longrightarrow < \mathrm{H}', \mathrm{Exc} >^{exc}}$$

The stack top element must be a reference of an object of type `Throwable`. If there is a handler that protects this bytecode instruction from the exception thrown, the control is transfered to the instruction at which the exception handler starts[3]. If the object on the stack top is `null` , a `NullPntrExc` is thrown.

[3]for every method the ExceptionHandler table describes the corresponding exception handler by the limits of the region it protects, the Exception that it catches, and the instruction at which it starts

- Method Invokation. invoke [4]

$$\frac{\begin{array}{l} meth :< \mathrm{H}, 0, [\,], [\mathrm{st}(\mathrm{cntr} - meth.nArgs), \ldots, \mathrm{st}(\mathrm{cntr})], 0 > \longrightarrow < \mathrm{H}', \mathrm{Res} >^{norm} \\ \Rightarrow \\ \begin{cases} \mathrm{cntr}' = \mathrm{cntr} - m.nArgs + 1 \\ \mathrm{st}\,' = \mathrm{st}\ [\oplus \mathrm{cntr}' \rightarrow \mathrm{Res}] \\ \mathrm{pc}\,' = \mathrm{pc}\ + 1 \end{cases} \end{array}}{m \vdash \ \mathrm{invoke}\ meth :< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ > \longrightarrow < \mathrm{H}', \mathrm{cntr}', \mathrm{st}\,', \mathrm{reg}\ , \mathrm{pc}\,' >}$$

$$\frac{\begin{array}{l} meth :< \mathrm{H}, 0, [\,], [\mathrm{st}(\mathrm{cntr} - meth.nArgs), \ldots, \mathrm{st}(\mathrm{cntr})], 0 > \longrightarrow < \mathrm{H}', \mathrm{Exc} >^{exc} \\ \Rightarrow \\ getStateOnExc\ (< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ >, typeof(\mathrm{Exc}), m.excHndls) = k \end{array}}{m \vdash \ \mathrm{invoke}\ meth :< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ > \longrightarrow k}$$

$$\frac{\begin{array}{l} \mathrm{st}(\mathrm{cntr} - meth.nArgs) == \texttt{null}\ \Rightarrow \\ getStateOnExc\ (< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ >,\ \texttt{NullPntrExc}, m.excHndls) = k \end{array}}{m \vdash \ \mathrm{invoke}\ meth :< \mathrm{H}, \mathrm{cntr}, \mathrm{st}\ , \mathrm{reg}\ , \mathrm{pc}\ > \longrightarrow k}$$

The first top $meth.nArgs$ elements in the operand stack st are popped from the operand stack. If st(cntr - meth.$nArgs$) is not `null` , the invoked method is executed on the object st(cntr - meth.$nArgs$) and where the first $nArgs+1$ elements of the list of its of local variables is initialised with st(cntr - meth.$nArgs$) ... st(cntr). In case that the execution of method $meth$ terminates normally, the return value Res of its execution is stored on the operand stack of the invoker. If the execution of of method $meth$ terminates because of an exception Exc, then the exception handler of the invoker is searched for a handler that can handle the exception. In case the object st(cntr - meth.$nArgs$) on which the method $meth$ must be called is `null` , a `NullPntrExc`is thrown.

3 Specification language for Java Bytecode programs

3.1 Introduction

This section presents a bytecode level specification language, called for short BCSL and a compiler from a subset of the high level Java specification language JML to BCSL. BCSL can express functional properties of Java bytecode programs in the form of method pre and postconditions, class and object invariants, assertions for particular program points like loop invariants. Before going further, we discuss what advocates the need of a low level specification. Traditionally, specification languages were tailored for high level languages. Source specification allows to express functional or security properties about a program

[4] only the case when the invoked method returns a value

and they are / can successfully be used for software audit and validation. Still, source specification in the context of mobile code does not help a lot for several reasons. First, the executable / interpreted code may not be accompanied by its specified source. Second, it is more reasonable for the code receiver to check the executable code than its source code, especially if he is not willing to trust the compiler. Third, if the client has complex requirements and even if the code respects them, in order to establish them, the code should be specified. Of course, for properties like well typedness this specification can be inferred but for more sophisticated policies, an automatic inference will not work. It is in this perspective, that we propose to make the Java bytecode benefit from the source specification by defining the BCSL language and a compiler from JML towards BCSL.

what kind of techniques: abstract interp ?

In what follows subsection 3.2 introduces the basic features of JML, subsection 3.3 gives the formal grammar of the specification language and subsection 3.4 describes the compilation process from JML to BCSL. The full specification of the new user defined Java attributes in which the JML specification is compiled is given in the appendix.

3.2 A quick overview of JML

JML [9] (short for Java Modeling Language) is a behavioral interface specification language tailored to Java applications. JML follows the design-by-contract approach (see [4]), where classes are annotated with class invariants and method pre- and postconditions. Specification inside methods is also possible; for example one can specify loop invariants, or assertions predicates that must hold at specific program points.

JML specifications are written as comments so they are not visible by Java compilers. The JML syntax is close to the Java syntax: JML extends Java with few keywords and operators. For introducing method precondition and postcondition one has to use the keywords `requires` and `ensures` respectively, `modifies` keyword is followed by all the locations that can be modified by the method, `loop_invariant`, not surprisingly, stands for loop invariants, `loop_modifies` keyword gives the locations modified by loop invariants etc. The latter is not standard in JML and is an extension introduced in [6]. Special JML operators are, for instance, `\result` which stands for the value that a method returns if it is not void, the `\old(expression)` operator designates the value of `expression` in the prestate of a method and is usually used in the method's postcondition. JML also allows the declaration of special JML variables, that are used only for specification purposes. These variables are declared in comments with the `model` modificator and may be used only in specification clauses.

JML can be used for either static checking of Java programs by tools such as JACK, the Loop tool, ESC/Java [11] or dynamic checking by tools such as the assertion checker jmlrac [7]. An overview of the JML tools can be found in [5].

Figure 2 gives an example of a Java class that models a list stored in a private array field. The method `replace` will search in the array for the first occurence of the object `obj1` passed as first argument and if found, it will be replaced with the object passed as second argument `obj2` and the method will return true; otherwise it returns false. The loop in the method body has an

invariant which states that all the elements of the list that are inspected up to now are different from the parameter object `obj1`. The loop specification also states that the local variable `i` and any element of the array field `list` may be modified in the loop.

```
public class ListArray {
  private Object[] list;

  //@requires list != null;
  //@ensures \result ==(\exists int i;
  //@ 0 <= i && i < list.length &&
  //@ \old(list[i]) == obj1 && list[i] == obj2);
  public boolean replace(Object obj1,Object obj2)
  {
    int i = 0;
    //@loop_modifies i, list[*];
    //@loop_invariant i <= list.length && i >=0
    //@  && (\forall int k;0 <= k && k < i ==>
    //@  list[k] != obj1);
    for (i = 0; i < list.length; i++ ) {
      if ( list[i] == obj1) {
        list[i] = obj2;
        return true;
      }
    }
    return false;
  }
}
```

Figure 2: class `ListArray` with JML annotations

3.3 Features of BCSL

BCSL corresponds to a representative subset of JML and is expressive enough for most purposes including the description of non trivial functional and security properties.

Specification clauses in BCSL that are taken from JML and inherit their semantics directly from JML include:

- class specification, i.e. class invariants and history constraints
- ghost variables, which are special specification variables not seen by the virtual machine, used by tools that support BCSL, e.g. a verification condition generator for Java bytecode. Those variables can be assigned by using the special BCSL operator set.
- method specification cases. Every method specification case specifies a method precondition, normal and exceptional postconditions, method frame conditions (the locations that may be modified by the method).

- inter method specification, for instance loop invariants
- predicates from first order logic
- expressions from the programming language, like field access expressions, local variables, etc.
- specification operators. For instance `\old(`$\mathcal{E}$`)` which is used in method postconditions and designates the value of the expression $\mathcal{E}$ in the prestate of a method, `\result` which stands for the value the method returns if it is not void

BCSL has few particular extra features that JML lacks :

- loop frame condition, which declares the locations that can be modified during a loop iteration. We were inspired for this by the JML extensions in JACK [6]
- stack expressions - cntr which stands for the stack counter and st(`ArithmeticExpr`) standing for a stack element at position `ArithmeticExpr` . These expressions are needed in BCSL as the Java Virtual Machine (JVM) is stack based.

The formal grammar of BCSL is given in Fig. 3

ClassSpec ::=	$\mid ClassInv\ \mathcal{F}^{bc}$	
	$\mid ClassHistoryConstr\ \mathcal{F}^{bc}$	
	$\mid declare$ ghost $JavaType\ Name$	
InterMethodSpec ::=	$\mid loopInvariant\ \mathcal{F}^{bc}$	
	$\mid loopModifies\ list\ loc$	
	$\mid loopDecreases\mathcal{E}$	
	$\mid assert\ \mathcal{F}^{bc}$	
	$\mid$ set $\mathcal{E}\ \mathcal{E}$	(sets to a new value a ghost variable)
MethodSpec ::=	$\mid$ SpecCase	
	$\mid$ SpecCase *also* MethodSpec	(specifies several specification cases)
SpecCase ::=	$\mid requires\ \mathcal{F}^{bc}$	
	$\mid modifies\ list\ loc$	(specifies the locations modified by a method)
	$\mid ensures\ \mathcal{F}^{bc}$	
	$\mid exsures\ (Exception\ exc)\ \mathcal{F}^{bc}$	(specifies what is the postcondition in case the method terminates with exception exc)
specExpr ::=	$\mathcal{E}$	
	$\mid$ `\typeof(`$\mathcal{E}$`)`	(returns the dynamic type of $\mathcal{E}$)
	$\mid$ `\type(`$ClassName$`)`	
	$\mid$ `\elemtype(`$\mathcal{E}$`)`	(returns the type of the elements of the array $\mathcal{E}$)
	$\mid$ `\old(`$\mathcal{E}$`)`	(used in method postcondition and stands for the value of $\mathcal{E}$ in the prestate of the method)
	$\mid$ `\result`	(stands for the value returned by the method in case the method is not void)
op ::=	$+ \mid - \mid * \mid div \mid rem \mid bitwise$	
$\mathcal{R}$ ::=	$\mid == \mid \neq \mid \leq \mid \leq \mid \geq \mid > \mid subtype$	
$\mathcal{F}^{bc}$ ::=	$\mid \mathcal{E}_1\ rel\ \mathcal{E}_2,\ rel \in \mathcal{R}$	
	$\mid$ `true`	
	$\mid$ `false`	
	$\mid not\ \mathcal{F}^{bc}$	
	$\mid \mathcal{F}^{bc} \wedge \mathcal{F}^{bc}$	
	$\mid \mathcal{F}^{bc} \vee \mathcal{F}^{bc}$	
	$\mid \mathcal{F}^{bc} \Rightarrow \mathcal{F}^{bc}$	
	$\mid \forall\ x : Values.(\mathcal{F}^{bc}(x))$	
	$\mid \exists\ x : Values(\mathcal{F}^{bc}(x))$	
Values ::=	$\mid i, i \in int\ literal$	
	$\mid R, R \in$ RefCl	

Figure 3: grammar of BCSL

3.4 Compiling JML into bytecode specification language

We now turn to explaining how JML specifications are compiled into user defined attributes for Java class files. Recall that a class file defines a single class or interface and contains information about the class name, interfaces implemented by the class, super class, methods and fields declared in the class and references. The Java Virtual Machine Specification (JVMS) [12] mandates that the class file contains data structure usually referred as the **constant_pool** table which is used to construct the runtime constant pool upon class or interface creation. The runtime constant pool serves for loading, linking and resolution of references used in the class. The JVMS allows to add to the class file user specific information([12], ch.4.7.1). This is done by defining user specific attributes (their structure is predefined by JVMS).

Thus the "JML compiler" [5] compiles the JML source specification into user defined attributes. The compilation process has three stages:

1. Compilation of the Java source file. This can be done by any Java compiler that supplies for every method in the generated class file the **Line_Number_Table** and **Local_Variable_Table** attributes. The presence in the Java class file format of these attribute is optional [12], yet almost all standard non optimizing compilers can generate these data. The **Line_Number_Table** describes the link between the source line and the bytecode of a method. The **Local_Variable_Table** describes the local variables that appear in a method. Those attributes are important for the next phase of the JML compilation.

2. Compilation of the JML specification from the source file and the resulting class file. In this phase, Java and JML source identifiers are linked with their identifiers on bytecode level, namely with the corresponding indexes either from the constant pool or the array of local variables described in the **Local_Variable_Table** attribute. If, in the JML specification a field identifier appears for which no constant pool (cp) index exists, it is added in the constant pool and the identifier in question is compiled to the new cp index. It is also in this phase that the specification parts like the loop invariants and the assertions which should hold at a certain point in the source program must be associated to the respective program point on bytecode level. The specification is compiled in binary form using tags in the standard way. The compilation of an expression is a tag followed by the compilation of its subexpressions.

   Another important issue in this stage of the JML compilation is how the type differences on source and bytecode level are treated. By type differences we refer to the fact that the JVM (Java Virtual Machine) does not provide direct support for intergral types like byte, short, char, neither for boolean. Those types are rather encoded as integers in the bytecode. Concretely, this means that if a Java source variable has a boolean type it will be compiled to a variable with an integer type. For instance, in the example for the method `isElem` and its specification in Fig.2 the postcondition states the equality between the JML expression `\result` and a

[5] Gary Leavens also calls his tool jmlc JML compiler, which transforms jml into runtime checks and thus generates input for the jmlrac tool

$$\texttt{\result} = 1$$

$$\Longleftrightarrow$$

$$\exists var(0).\left(\begin{array}{l} 0 \leq var(0) \wedge \\ var(0) < len(\#19(reg_0\ )) \wedge \\ \#19(reg_0\ )[var(0)] = reg_1 \end{array}\right)$$

Figure 4: The compilation of the postcondition in Fig. 2

predicate. This is correct as the method `isElem` in the Java source is declared with return type boolean and thus, the expression `\result` has type boolean. Still, the bytecode resulting from the compilation of the method `isElem` returns a value of type integer. This means that the JML compiler has to "make more effort" than simply compiling the left and right side of the equality in the postcondition, otherwise its compilation will not make sense as it will not be well typed. Actually, if the JML specification contains program boolean expressions that the Java compiler will compile to bytecode expression with an integer type, the JML compiler will also compile them in integer expressions and will transform the specification condition in equivalent one[6].

Finally, the compilation of the postcondition of method `isElem` is given in Fig. 4. From the postcondition compilation, one can see that the expression `\result` has integer type and the equality between the boolean expressions in the postcondition in Fig.2 is compiled into logical equivalence. The example also shows that local variables and fields are respectively linked to the index of the register table for the method and to the corresponding index of the constant pool table (#19 is the compilation of the field name `list` and reg_1 stands for the method parameter `obj`).

3. add the result of the JML compilation in the class file as user defined attributes. Method specifications, class invariants, loop invariants are newly defined attributes in the class file. For example, the specifications of all the loops in a method are compiled to a unique method attribute: whose syntax is given in Fig. 5. This attribute is an array of data structures each describing a single loop from the method source code. Also for each loop in the source code there must be a corresponding element in the array. More precisely, every element contains information about the instruction where the loop starts as specified in the **Line_Number_Table**, the locations that can be modified in a loop iteration, the invariant associated to this loop and the decreasing expression in case of total correctness,

The JML compiler does not depend on any specific Java compiler, but it requires the presence of a debugging information, namely the presence of the **Line_Number_Table** attribute for the correct compilation of inter method specification, i.e. loops and assertions. We think that this is an acceptable restriction for the compiler. The most problematic part of the compilation is

[6] when generating proof obligations we add for every source boolean expression an assumption that it must be equal to 0 or 1. Actually, a reasonable compiler will encode boolean values in this way

```
JMLLoop_specification_attribute {
    ...
    {  u2 index;
       u2 modifies_count;
       formula modifies[modifies_count];
       formula invariant;
       expression decreases;
    } loop[loop_count];
}
```

- **index**: The index in the `LineNumberTable` where the beginning of the corresponding loop is described
- **modifies**[]: The array of locations that may be modified
- **invariant** : The predicate that is the loop invariant. It is a compilation of the JML formula in the low level specification language
- **decreases**: The expression which decreases at every loop iteration

Figure 5: Structure of the Loop Attribute

to identify which source loop corresponds to which bytecode loop in the control flow graph. To do this, we assume that the control flow graph is reducible (see [1]), i.e. there are no jumps from outside a loop inside it; graph reducibility allows to establish the same order between loops in the bytecode and source code level and to compile the invariants to the correct places in the bytecode.

4 Weakest Precondition Calculus for Java Bytecode

This section describes the weakest precondition predicate transformer function which underlines the verification condition generator which will be presented later. The calculus is defined for the full Java sequential subset except for 64 bit data and floating point arithmetic and thus, deals with object manipulation and creation, exception throwing and handling, subroutines. The weakest precondition predicate transformer understands the specification language BCSL.

4.1 Bytecode Programs. Background

In the following, we give a background information about programs written in our bytecode language and their execution. A Java bytecode program is a set of Java classes. Every Java class consists of a set of method declaration. Every method declaration (if not an abstract or interface method) has an array `instr` of bytecode instructions which constitute the method body. The $k-th$ instruction in the bytecode array `instr` is denoted with $\texttt{instr}_k$. We assume that the method bytecode has exactly one entry point (an entry point instruction is the instruction at which an execution of a method starts) and we denote it

with *entryPnt*. If there is an execution path in which the instruction $\texttt{instr}_j$ may execute immediately after $\texttt{instr}_k$, we note it with $\texttt{instr}_k \rightarrow \texttt{instr}_j$.

We assume that the control flow graph is reducible, i.e. every loop has exactly one entry point. This actually is admissible as it is rarely the case that a compiler produce a bytecode with a non reducible flow graph and the practice shows that even hand written code is usually reducible. Anyways, there are algorithms to transform a non reducible control flow graph to a reducible one. For more information on reducible flow graph, one can look at [1]. The next definition identifies backedges in the reducible control flow graph (intuitively, the edge that goes from from an instruction in a given loop in the control flow graph to the loop entry) with the special execution relation $\rightarrow^l$ as follows:

Definition 1 (Loop Definition) *Let's have a bytecode program* Π. *We say that* $\texttt{instr}_e$ *is the entry instruction of loop* l *and* $\texttt{instr}_f$ *is the end instruction of* l *if and we note with* $\texttt{instr}_f \rightarrow^l \texttt{instr}_e$ *:*

- *every path in the control flow graph starting at the entry point entryPnt that reaches* $\texttt{instr}_f$*, passes through* $\texttt{instr}_e$
- *there is a path in which* $\texttt{instr}_e$ *is executed immediately after the execution of* $\texttt{instr}_f$ *(* $\texttt{instr}_e \rightarrow \texttt{instr}_f$*)*

We illustrate the upper definitions with the control flow graph of the example from Fig.2 in Fig. 6. In the figure we rather show the execution relation between basic blocks (a standard notion denoting a sequence of instructions where only the last one may be a jump and the first may be a target of a jump) the execution relation between the intstructions in a block being evident.

4.2 WP for a Java bytecode

In what follows we assume that the bytecode has passed the bytecode verifier, thus it is well typed and well structured. Actually, our calculus is concerned with the program functional properties leaving the problem of code well structuredness and welltypedness to the bytecode verification techniques

4.2.1 Definitions

The predicate transformer function which for any instruction of the Java sequential fragment, depending on the position of the instruction and on the function ψ_{exc} determines the predicate that must hold in the prestate of the instruction has the following signature :

$$\text{wp} : \texttt{instr} \longrightarrow \texttt{int} \longrightarrow (\texttt{int} \longrightarrow \texttt{Exception} \longrightarrow \texttt{Predicate}) \longrightarrow \texttt{Predicate}$$

The function defined as follows :

$$\texttt{exc}\ \text{m}\ : \texttt{Exception} \longrightarrow \texttt{Predicate}$$

returns the predicate that must hold if the method m ends by throwing an exception

The function $\psi_{exc}(\texttt{i}, \texttt{Exc})$ where i is an index in the bytecode of method m is defined as follows:

- it returns precondition of the exception handler, if an exception handler that protects the instruction at index i from exception of type `Exc` exists
- otherwise it is equal to `exc` m (`Exc`)

Also, every time an exception of type `E` is thrown a new object of type `E` is allocated on the heap $\mathrm{H} \oplus [\ \mathtt{ref}_E \longrightarrow \mathtt{obj}(\mathtt{E}, \mathtt{f})]$ and the reference $\mathtt{ref}_E$ is put on the stack top.

We denote the postcondition of a method with ψ^n. *Note:* In what follows we ignore the Java error exceptions. Thus we deal only with user defined exceptions and `JavaRuntimeException` subclasses

First we define the function *inter* that for two instructions that may execute one after another in some execution graph determines the predicate $\mathrm{inter}((, k), j)$ the predicate that must hold in between them. This definition determines if at a given program point there is an invariant to hold or it is the precondition of the next instruction to hold.

Definition 2 (intermediate predicate between two instructions) *Assume that* $\mathtt{instr}_k \rightarrow \mathtt{instr}_j$. *The predicate* $\mathrm{inter}(k, j)$ *must hold after the execution of* $\mathtt{instr}_k$ *and before the execution of* $\mathtt{instr}_j$ *and is defined as follows:*

- *if* $\mathtt{instr}_k \rightarrow^l \mathtt{instr}_j$ *then the corresponding loop invariant must hold:*

$$\mathrm{inter}(k, j) = I$$

- *else if* $\mathtt{instr}_j$ *is a loop entry then the corresponding loop invariant* I *must hold before* $\mathtt{instr}_j$ *is executed, i.e. after the execution of* $\mathtt{instr}_k$. *We also require that* I *implies the weakest precondition of the loop entry instruction. The implication is quantified over the locations* $m_i, i = 1..s$ *that may be modified in the loop.*

$$\mathrm{inter}(k, j) = I \ \wedge \ \forall_{i=1..s} m_i.(I \Rightarrow \ \mathrm{wp}\ (\ \mathtt{instr}_\mathrm{j}, \mathrm{j}, \psi_{\mathrm{exc}}))$$

- *else*

$$\mathrm{inter}(k, j) = \ \mathrm{wp}\ (\ \mathtt{instr}_\mathrm{j}, \mathrm{j}, \psi_{\mathrm{exc}})$$

4.2.2 Weakest Precondition Calculus Rules

We now define the rules for the predicate transfomer wp for our bytecode programming language

- Control transfer instructions
  1. unconditional jumps

$$\mathrm{wp}\ (\ \mathrm{goto}\ \ \mathrm{n}, \mathrm{i}, \psi_{\mathrm{exc}}) = \ \mathrm{inter}(\mathrm{i}, \mathrm{n})$$

  2. conditional jumps

     $\mathrm{wp}\ (\ \mathrm{if\_cond}\ \ \mathrm{n}, \mathrm{i}, \psi_{\mathrm{exc}}) =$

$$\begin{cases} \mathtt{cond}(\mathrm{st}(\mathrm{cntr}), \mathrm{st}(\mathrm{cntr}-1)) \Rightarrow \ \ \mathrm{inter}(i, n)[\mathrm{cntr} \leftarrow \mathrm{cntr}-2] \\ \wedge \\ \mathit{not}(\mathtt{cond}(\mathrm{st}(\mathrm{cntr}), \mathrm{st}(\mathrm{cntr}-1))) \Rightarrow \ \ \mathrm{inter}(i, i+1)[\mathrm{cntr} \leftarrow \mathrm{cntr}-2] \end{cases}$$

3. return
   wp (return, i, ψ_{exc}) = ψ^{n}[\result ← st(cntr)]

- subroutines
  Subroutines are treated by inlining, thus every the precondition of every jsr instruction depends on the subroutine code which is executed after it.

  1. wp (jsr n, i, ψ_{exc}) = inter(i, n)

  2. wp (ret n, i, ψ_{exc}) = inter(i, k) for any instruction $\texttt{instr}_k$ which follows a `jsr` instruction, that jumps to the subroutine ending with $\texttt{instr}_i$

- load and store instructions

  1. load
     wp (load j, i, ψ_{exc}) =
     inter$(i, i+1)$[cntr ← cntr + 1][st(cntr + 1) ← reg_j]

  2. store
     wp (store j, i, ψ_{exc}) =
     inter$(i, i+1)$[cntr ← cntr − 1][reg_j ← st(cntr)]

  3. push
     wp (push j, i, ψ_{exc}) =
     inter$(i, i+1)$[cntr ← cntr + 1][$signExtendToInt(j)$ ← st(cntr + 1)]

     The value `j`, which is of type `byte` in the case push = bipush and is of type `short` in case push = sipush . It is sign extended to int before pushed on the stack.

  4. iinc wp (iinc j, i, ψ_{exc}) =
     inter$(i, i+1)$[reg_j ← reg_j + 1]

- arithmetic instructions

  1. instruction that cannot cause exception throwing (`arithOp` = add , sub , mult , and , or , xor , ishr , ishl ,)
     wp (arithOp, i, ψ_{exc}) =
     inter$(i, i+1)$[cntr ← cntr − 1][st(cntr − 1) ← st(cntr) `op` st(cntr − 1)]
  2. instructions that may throw exceptions (`arithOp` = rem , div)
     wp (arithOp , i, ψ_{exc}) =

$$
\begin{cases}
\text{st(cntr)} \neq \texttt{null} \;\Rightarrow \\
\quad \text{inter}(i, i+1)[\text{cntr} \leftarrow \text{cntr} - 1] \\
\quad\quad [\text{st(cntr} - 1) \leftarrow \text{st(cntr)}\ \texttt{op}\ \text{st(cntr} - 1)] \\
\wedge \\
\text{st(cntr)} = \texttt{null} \;\Rightarrow \\
\quad \psi_{exc}(i, \texttt{ArithmeticExc})[\text{cntr} \leftarrow \texttt{0}] \\
\quad [\text{H} \leftarrow \text{H}[\oplus\ \texttt{ref}_{ArithmeticExc} \longrightarrow \texttt{Obj}_{\texttt{ArithmeticExc}}]] \\
\quad [\text{st(0)} \leftarrow\ \texttt{ref}_{ArithmeticExc}]
\end{cases}
$$

- object creation and manipulation

  1. new
     wp (new Class, i, ψ_{exc}) =

     $$\begin{array}{l} \text{inter}(i, i+1)[\text{cntr} \leftarrow \text{cntr}+1] \\ \quad [\text{st}(\text{cntr}+1) \leftarrow \texttt{ref}_{Class}] \\ \quad [\text{H} \leftarrow \text{H}[\oplus\ \texttt{ref}_{Class} \longrightarrow \texttt{Obj}_{\texttt{Class}}]] \end{array}$$

     H is updated with the newly created object and a reference to it is put on the top of the stack:

  2. array creation
     wp (newarray type, i, ψ_{exc}) =

     $$\begin{array}{l} \text{st}(\text{cntr}) \geq 0 \Rightarrow \\ \qquad \text{inter}(i, i+1)[\text{H} \leftarrow \text{H}[\oplus\ \texttt{ref}_{type} \longrightarrow \texttt{Obj}_{\texttt{type},\text{st}(\text{cntr})}] \\ \quad [\text{st}(\text{cntr}) \leftarrow \texttt{ref}_{type}] \\ \wedge \\ \text{st}(\text{cntr}) < 0 \Rightarrow \\ \qquad \psi_{exc}(i, \texttt{NegativeArraySizeExc}) \\ \qquad [\text{H} \leftarrow \text{H}[\oplus\ \texttt{ref}_{NegativeArraySizeExc} \longrightarrow \texttt{Obj}_{\texttt{NegativeArraySizeExc}}]] \\ \qquad [\text{cntr} \leftarrow 0] \\ \qquad [\text{st}(0) \leftarrow \texttt{ref}_{NegativeArraySizeExc}] \end{array}$$

     where a newly created object `obj(type,` st`(` cntr `)` is allocated in the H and a reference $\texttt{ref}_{type}$ to it is put on the stack top H = H[$\oplus$ $\texttt{ref}_{type} \longrightarrow$ `obj(type,` st`(` cntr `)`]

  3. getfield
     wp (getField f , i, ψ_{exc}) =

     $$\begin{array}{l} \text{st}(\text{cntr}) \neq \texttt{null} \Rightarrow \\ \quad \text{inter}(i, i+1)[\text{st}(\text{cntr}) \leftarrow \texttt{f}(\text{st}(\text{cntr}))] \\ \wedge \\ \text{st}(\text{cntr}) = \texttt{null} \Rightarrow \\ \qquad \psi_{exc}(i, \texttt{NullPointerExc}) \\ \qquad\quad [\text{H} \leftarrow \text{H}[\oplus\ \texttt{ref}_{NullPointerExc} \longrightarrow \texttt{Obj}_{\texttt{NullPointerExc}}]] \\ \qquad [\text{cntr} \leftarrow 0] \\ \qquad\quad [\text{st}(0) \leftarrow \texttt{ref}_{NullPointerExc}] \end{array}$$

  4. putfield
     wp (putField f , i, ψ_{exc}) =

     $$\begin{array}{l} \text{st}(\text{cntr}) \neq \texttt{null} \Rightarrow \\ \quad \text{inter}(i, i+1)[\text{cntr} \leftarrow \text{cntr}-2] \\ \qquad [\texttt{f} \leftarrow \texttt{f}(\oplus[\text{st}(\text{cntr}-\texttt{2}) \longrightarrow \text{st}(\text{cntr}-\texttt{1})])] \\ \wedge \\ \text{st}(\text{cntr}) = \texttt{null} \Rightarrow \\ \qquad \psi_{exc}(i, \texttt{NullPointerExc}) \\ \qquad\quad [\text{H} \leftarrow \text{H}[\oplus\ \texttt{ref}_{NullPointerExc} \longrightarrow \texttt{Obj}_{\texttt{NullPointerExc}}]] \\ \qquad\quad [\text{cntr} \leftarrow 0] \\ \qquad\quad [\text{st}(0) \leftarrow \texttt{ref}_{NullPointerExc}] \end{array}$$

5. arraylength - returns the length of the array which is on the top of the stack
   wp (arraylength f , i, ψ_{exc}) =

$$
\begin{array}{l}
\text{st(cntr)} \neq \texttt{null} \ \Rightarrow \\
\ \text{inter}(i, i+1) \quad [\text{st(cntr)} \leftarrow \mathit{length}(\text{st(cntr)})] \\
\wedge \\
\text{st(cntr)} = \texttt{null} \ \Rightarrow \\
\qquad \psi_{exc}(i, \texttt{NullPointerExc}) \\
\qquad\quad [\text{cntr} \leftarrow 0] \\
\qquad\quad [\text{st}(0) \leftarrow \ \texttt{ref}_{NullPointerExc}]
\end{array}
$$

6. checkcast wp (checkcast Class, i, ψ_{exc}) =

$$
\begin{array}{l}
\mathit{subtype}(\mathit{getType}(\text{st(cntr)}), \texttt{Class}) \vee \text{st(cntr)} = \texttt{null} \ \Rightarrow \\
\ \text{inter}(i, i+1) \\
\wedge \\
\mathit{not}(\mathit{subtype}(\mathit{getType}(\text{st(cntr)}), \texttt{Class})) \Rightarrow \\
\qquad \psi_{exc}(i, \texttt{ClassCastExc}) \\
\qquad\quad [\text{H} \leftarrow \text{H}[\oplus \ \texttt{ref}_{NullPointerExc} \longrightarrow \texttt{Obj}_{\texttt{NullPointerExc}}]] \\
\qquad\quad [\text{cntr} \leftarrow 0] \\
\qquad\quad [\text{st(cntr)} \leftarrow \ \texttt{ref}_{ClassCastExc}]
\end{array}
$$

7. instanceof wp (instanceof Class, i, ψ_{exc}) =

$$
\begin{array}{l}
\mathit{subtype}(\mathit{getType}(\text{st(cntr)}), \texttt{Class}) \Rightarrow \\
\ \text{inter}(i, i+1)[\text{st(cntr)} \leftarrow 1] \\
\wedge \\
\mathit{not}(\mathit{subtype}(\mathit{getType}(\text{st(cntr)}), \texttt{Class})) \vee \text{st(cntr)} = \texttt{null} \ \Rightarrow \\
\ \text{inter}(i, i+1)[\text{st(cntr)} \leftarrow 0]
\end{array}
$$

- method invocation (only the case for non void instance method is given). We take into account the normal and abnormal termination cases for the method invocation. The rules states that the precondition of the called method must be established; the postcondition of the called method is assumed to hold.
  wp (invoke m, i, ψ_{exc}) =

$$
\begin{array}{l}
\mathit{pre}(\texttt{m})[\mathit{reg}_s \ \leftarrow \text{st}(\text{cntr} + s - \mathit{numargs}(m))]_{s=0}^{\mathit{numargs}(m)} \\
\wedge \\
\forall \texttt{e}_{\texttt{m}}(\texttt{m} = \texttt{1..k})(\\
\mathit{post}(\texttt{m})[\backslash\texttt{result} \leftarrow \mathit{freshVar}] \\
\qquad [\mathit{reg}_s \ \leftarrow \text{st}(\text{cntr} + s - \mathit{numargs}(m))]_{s=0}^{\mathit{numargs}(m)} \\
\qquad [\text{H} \leftarrow \text{H}[\oplus \ \texttt{ref}_{Cl_s} \longrightarrow \texttt{Obj}_{\texttt{Cl}_\texttt{s}}]_{\texttt{s=0}}^{\texttt{t}}] \\
\qquad \Rightarrow \\
\qquad\quad \text{inter}(i, i+1)[\text{cntr} \leftarrow \text{cntr} - \texttt{numargs(m)}] \\
\qquad\quad\ [\text{st}(\text{cntr} - \mathit{numargs}(m)) \leftarrow \mathit{freshVar}]) \\
\wedge_{j=1}^{s} \\
\forall \texttt{e}_{\texttt{m}}(\texttt{m} = \texttt{1..k})(\psi_{\texttt{exc}}^{\texttt{m}}(\texttt{Exc}_\texttt{j}) \Rightarrow \psi_{\texttt{exc}}(\texttt{i}, \texttt{Exc}_\texttt{j}))
\end{array}
$$

  $\texttt{Exc}_\texttt{j}, \texttt{j} = \texttt{1..s}$ the exceptions that the method m may throw

$\psi^{\mathtt{m}}_{exc}$ the function which returns the predicate $\psi^{\mathtt{m}}_{exc}(\mathtt{Exc})$ that holds if the exception `Exc` is thrown.

$\psi_{exc}(i,\ \mathtt{ref}_{Exc_j})$ is the either the precondition of the exception handler protecting from exceptions of type $\mathtt{Exc_j}$ at index i or is the exceptional postcondition $excPost(\mathtt{Exc_j})$ if such an exception handler does not exist

$pre(\mathtt{m})$ - the precondition of method `m`

$post(\mathtt{m})$ - the postcondition of method `m`

$\mathtt{e_m(m = 1..k)}$ - the locations that may be modified by method `m`

also the rule reflects that in the heap may have been allocated new objects during the execution of the invoked method

- throw exception instruction

wp (athrow , i, ψ_{exc}) =

$$\begin{array}{l} \text{st(cntr)} \neq \mathtt{null} \ \Rightarrow \psi_{exc}(i, \text{st(cntr)}) \\ \wedge \\ \text{st(cntr)} = \mathtt{null} \ \Rightarrow \psi_{exc}(i,\ \mathtt{ref}_{NullPointerExc}) \end{array}$$

The thrown object is on the top of the stack, and that is why the precondition of the athrow instruction depends on st(cntr).

4.3 Example

4.3.1 Weakest Precondition in the Presence of Loop

The following example shows how the weakest precondition calculus is applied to the example program for which we have shown the source code at Fig. 2 and its bytecode control flow graph at Fig. 6. In particular, the example that comes here after gives the weakest preconditions for the instructions that are part of the loop of the method body. Thus, every instruction is preceded with its weakest precondition enclosed in curly braces. The invariant is written at the place where it must hold (i.e. where the backedge relating the loop entry and the loop end is). The weakest precondition of every instruction is calculated w.r.t. the weakest precondition of the next instruction in the execution path. So, looking at the example, the precondition of instruction 19 is calculated w.r.t. the loop invariant. The weakest precondition of the instruction 18 is calculated w.r.t. the wp of instruction 19 etc. Also the wp of the jump instruction 24 is calculated from the weakest precondition of instruction 6. This example shows the process of calculating the verification conditions for a program.

The wp of a method body is the conjunction the wps of all of its execution paths.

...

{
$\text{reg}_0 = \texttt{null} \Rightarrow \texttt{false} \wedge$
$\text{reg}_0 \neq \texttt{null} \Rightarrow$
$\quad test.ListArray.list(\text{reg}_0) \neq \texttt{null} \wedge$
$\quad len(test.ListArray.list(\text{reg}_0)) > \text{reg}_3 \wedge$
$\quad \text{reg}_3 \geq 0 \wedge$
$\quad \text{reg}_1 \neq \text{ListArray.list}(\text{reg}_0)[\text{reg}_3] \Rightarrow$
$\quad\quad 1 + \text{reg}_3 \leq \text{len}(\text{test.ListArray.list}(\text{reg}_0)) \wedge$
$\quad\quad 1 + \text{reg}_3 \geq 0 \wedge$
$\quad\quad \forall var(0).0 \leq var(0) \wedge var(0) < 1 + \text{reg}_3 \Rightarrow$
$\quad\quad\quad ListArray.list(\text{reg}_0)[\text{var}(0)] \neq \text{reg}_1$
$\wedge$
$ListArray.list(\text{reg}_0) = \texttt{null} \Rightarrow \texttt{false} \wedge$
$len(test.ListArray.list(\text{reg}_0)) \leq \text{reg}_3 \vee \text{reg}_3 < 0 \Rightarrow \texttt{false}$
}

6 aload_0

{
$\text{st(cntr)} = \texttt{null} \Rightarrow \texttt{false} \wedge$
$\text{st(cntr)} \neq \texttt{null} \Rightarrow$
$\quad ListArray.list(\text{st(cntr)}) \neq \texttt{null} \wedge$
$\quad len(test.ListArray.list(\text{st(cntr)})) > \text{reg}_3 \wedge$
$\quad \text{reg}_3 >= 0 \wedge$
$\quad \text{reg}_1 \neq \text{ListArray.list}(\text{st(cntr)})[\text{reg}_3] \Rightarrow$
$\quad\quad 1 + \text{reg}_3 <= \text{len}(\text{test.ListArray.list}(\text{reg}_0)) \wedge$
$\quad\quad 1 + \text{reg}_3 >= 0 \wedge$
$\quad\quad \forall var(0).\ 0 \leq var(0) \wedge var(0) < 1 + \text{reg}_3 \Rightarrow$
$\quad\quad\quad ListArray.list(\text{reg}_0)[\text{var}(0)] \neq \text{reg}_1 \wedge$
$ListArray.list(\text{st(cntr)}) = \texttt{null} \Rightarrow \texttt{false} \wedge$
$len(test.ListArray.list(\text{st(cntr)})) <= \text{reg}_3 \vee \text{reg}_3 < 0 \Rightarrow \texttt{false}$
}

7 getfield ListArray.list

{
st(cntr) $\neq$ `null` $\wedge$
len(st(cntr)) > reg$_3\wedge$
reg$_3 \geq 0\wedge$
reg$_1 \neq$ st(cntr)[reg$_3$] $\Rightarrow$
    $1 +$ reg$_3 \leq$ len(ListArray.list(reg$_0$))$\wedge$
    $1 +$ reg$_3 >= 0\wedge$
    $\forall var(0).0 <= var(0) \wedge (var(0) < 1 +$ reg$_3 \Rightarrow$
        $ListArray.list$(reg$_0$)[var(0)] $\neq$ reg$_1\wedge$
st(cntr) = `null`) $\Rightarrow$ `false`$\wedge$
len(st(cntr)) $\leq$ reg$_3 \vee$ reg$_3 < 0 \Rightarrow$ `false`
}
8 iload_3

{
st(cntr $-$ 1) $\neq$ `null` $\wedge$
len(st(cntr $-$ 1) > st(cntr)$\wedge$
st(cntr) $\geq 0)\wedge$
reg$_1 \neq$ st(cntr $-$ 1)[st(cntr)] $\Rightarrow$
    $1 +$ reg$_3 \leq$ len(test.ListArray.list(reg$_0$))$\wedge$
    $1 +$ reg$_3 \geq 0\wedge$
    $\forall var(0).\ 0 \leq var(0) \wedge var(0) < 1 +$ reg$_3 \Rightarrow$
            $ListArray.list$(reg$_0$)[var(0)] $\neq$ reg$_1\wedge$
st(cntr $-$ 1) = `null`) $\Rightarrow$ `false`$\wedge$
len(st(cntr $-$ 1)) $\leq$ st(cntr) $\vee$ st(cntr) $< 0 \Rightarrow$ `false`
}
9 aaload

{
reg$_1 \neq$ st(cntr) $\Rightarrow$
    $1 +$ reg$_3 \leq$ len(test.ListArray.list(reg$_0$)$\wedge$
    $1 +$ reg$_3 \geq 0\wedge$
    $\forall var(0).\ 0 \leq var(0) \wedge var(0) < 1 +$ reg$_3 \Rightarrow$ ListArray.list(reg$_0$)[var(0)] $\neq$ reg$_1$
}
10 aload_1

{
st(cntr) $\neq$ st(cntr $-$ 1) $\Rightarrow$
    $1 +$ reg$_3 \leq$ len(test.ListArray.list(reg$_0$))$\wedge$
    $1 +$ reg$_3 \geq 0\wedge$
    $\forall var(0).\ 0 \leq var(0) \wedge var(0) < 1 +$ reg$_3 \Rightarrow$
        $ListArray.list$(reg$_0$)[var(0)] $\neq$ reg$_1$
}
11 if_acmpne19

...
...

{
$1 + \text{reg}_3 \leq \text{len}(\text{test.ListArray.list}(\text{reg}_0)))\wedge$
$1 + \text{reg}_3 \geq 0\wedge$
$\forall var(0).\ 0 \leq var(0) \wedge var(0) < 1 + \text{reg}_3 \Rightarrow$
    $ListArray.list(\text{reg}_0)[\text{var}(0)] \neq \text{reg}_1$
}
19 iinc 3

{
invariant :
$\text{reg}_3 \leq \text{len}(\text{test.ListArray.list}(\text{reg}_0))\wedge$
$\text{reg}_3 \geq 0\wedge$
$\forall var(0).\ 0 \leq var(0) \wedge var(0) < \text{reg}_3 \Rightarrow$
    $ListArray.list(\text{reg}_0)[\text{var}(0)] \neq \text{reg}_1$
}

{
$\forall var(27).\ var(27) \neq \text{reg}_(0)\wedge$
$\forall var(26).\ var(26) \geq 0 \wedge var(26) < len(test.ListArray.list(var(27)) \Rightarrow$
    $ListArray.list(var(27))[var(26)] = ListArray.list\_at\_20(var(27))[var(26)]\wedge$
    $\text{reg}_3\text{\_at\_20} \leq \text{len}(\text{ListArray.list\_at\_20}(\text{reg}_0))\wedge$
    $\text{reg}_3\text{\_at\_20} \geq 0\wedge$
    $\forall var(0).\ 0 \leq var(0) \wedge var(0) < \text{reg}_3\text{\_at\_20} \Rightarrow$
        $ListArray.list\_at\_20(\text{reg}_0)[\text{var}(0)] \neq \text{reg}_1$
$\Rightarrow$
    $\text{reg}_0 = \texttt{null}\ \Rightarrow false\wedge$
    $\text{reg}_0 \neq \texttt{null}\ \Rightarrow$
        $ListArray.list\_at\_20(\text{reg}_0) \neq \texttt{null}\ \wedge$
        $\text{reg}_3\text{\_at\_20}) < \text{len}(\text{ListArray.list\_at\_20}(\text{reg}_0)) \Rightarrow$
            $\text{reg}_0 = \texttt{null}\ \Rightarrow false\wedge$
            $\text{reg}_0 \neq \texttt{null}\ \Rightarrow$
                $ListArray.list\_at\_20(\text{reg}_0) \neq \texttt{null}\ \wedge$
                $len(ListArray.list\_at\_20(\text{reg}_0)) > \text{reg}_3\text{\_at\_20}\wedge$
                $\text{reg}_3\text{\_at\_20} \geq 0\wedge$
                $\text{reg}_1 \neq \text{ListArray.list\_at\_20}(\text{reg}_0)[\text{reg}_3\text{\_at\_20}] \Rightarrow$
                    $1 + \text{reg}_3\text{\_at\_20} \leq \text{len}(\text{test.ListArray.list\_at\_20}(\text{reg}_0))\wedge$
                    $1 + \text{reg}_3\text{\_at\_20} \geq 0\wedge$
                    $\forall var(0).0 \leq var(0)) \wedge var(0) < 1 + \text{reg}_3\text{\_at\_20} \Rightarrow$
                        $ListArray.list\_at\_20)(\text{reg}_0)[\text{var}(0)] \neq \text{reg}_1\wedge$
                    $ListArray.list\_at\_20(\text{reg}_0) = \texttt{null}\ \Rightarrow false\wedge$
                    $len(test.ListArray.list\_at\_20)(\text{reg}_0) \leq \text{reg}_3\text{\_at\_20} \vee \text{reg}_3\text{\_at\_20} < 0 \Rightarrow false))$
}
20 aload_3

{
$\mathrm{reg}_0 = \texttt{null} \Rightarrow \mathit{false} \wedge$
$\mathrm{reg}_0 \neq \texttt{null} \Rightarrow$
$\mathit{ListArray.list}(\mathrm{reg}_0) \neq \texttt{null} \wedge$
$\mathrm{st}(\mathrm{cntr} - 1) < \mathit{len}(\mathit{test.ListArray.list}(\mathrm{reg}_0) \Rightarrow$
$\mathrm{reg}_0 = \texttt{null} \Rightarrow \mathit{false} \wedge$
$\mathrm{reg}_0 \neq \texttt{null} \Rightarrow$
$\mathit{ListArray.list}(\mathrm{reg}_0) \neq \texttt{null} \wedge$
$\mathit{len}(\mathit{test.ListArray.list}(\mathrm{reg}_0)) > \mathrm{reg}_3 \wedge$
$\mathrm{reg}_3 \geq 0) \wedge$
$\mathrm{reg}_1 \neq \mathrm{ListArray.list}(\mathrm{reg}_0)[\mathrm{reg}_3] \Rightarrow$
$1 + \mathrm{reg}_3 \leq \mathrm{len}(\mathrm{test.ListArray.list}(\mathrm{reg}_0)) \wedge$
$1 + \mathrm{reg}_3 \geq 0 \wedge$
$\forall \mathit{var}(0).\ 0 \leq \mathit{var}(0) \wedge \mathit{var}(0) < 1 + \mathrm{reg}_3 \Rightarrow$
$\mathit{ListArray.list}(\mathrm{reg}_0)[\mathrm{var}(0)] \neq \mathrm{reg}_1))) \wedge$
$\mathit{ListArray.list}(\mathrm{reg}_0) = \texttt{null} \Rightarrow \mathit{false} \wedge$
$\mathit{len}(\mathit{test.ListArray.list}(\mathrm{reg}_0)) \leq \mathrm{reg}_3) \vee (\mathrm{reg}_3 < 0) \Rightarrow \mathit{false})) \wedge$
$\mathit{ListArray.list}(\mathrm{reg}_0) = \texttt{null}\ ) \Rightarrow \mathit{false}$
}
21 aload_0

{
$\mathrm{st}(\mathrm{cntr}) = \texttt{null} \Rightarrow \mathit{false} \wedge$
$\mathrm{st}(\mathrm{cntr}) \neq \texttt{null} \Rightarrow$
$\mathit{ListArray.list}(\mathrm{st}(\mathrm{cntr})) \neq \texttt{null} \wedge$
$\mathrm{st}(\mathrm{cntr} - 1) < \mathit{len}(\mathit{test.ListArray.list}(\mathrm{st}(\mathrm{cntr})) \Rightarrow$
$\mathrm{reg}_0 = \texttt{null} \Rightarrow \mathit{false} \wedge$
$\mathrm{reg}_0 \neq \texttt{null} \Rightarrow$
$\mathit{ListArray.list}(\mathrm{reg}_0) \neq \texttt{null} \wedge$
$\mathit{len}(\mathit{test.ListArray.list}(\mathrm{reg}_0)) > \mathrm{reg}_3 \wedge$
$\mathrm{reg}_3 \geq 0) \wedge$
$\mathrm{reg}_1 \neq \mathrm{ListArray.list}(\mathrm{reg}_0)[\mathrm{reg}_3] \Rightarrow$
$1 + \mathrm{reg}_3 \leq \mathrm{len}(\mathrm{test.ListArray.list}(\mathrm{reg}_0)) \wedge$
$1 + \mathrm{reg}_3 \geq 0 \wedge$
$\forall \mathit{var}(0).\ (((0 \leq \mathit{var}(0)) \wedge (\mathit{var}(0) < (1 + \mathrm{reg}_3))) \Rightarrow$
$\mathit{ListArray.list}(\mathrm{reg}_0)[\mathrm{var}(0)] \neq \mathrm{reg}_1))) \wedge$
$\mathit{ListArray.list}(\mathrm{reg}_0) = \texttt{null} \Rightarrow \mathit{false} \wedge$
$\mathit{len}(\mathit{test.ListArray.list}(\mathrm{reg}_0)) \leq \mathrm{reg}_3) \vee (\mathrm{reg}_3 < 0) \Rightarrow \mathit{false})) \wedge$
$\mathit{ListArray.list}(\mathrm{st}(\mathrm{cntr})) = \texttt{null}\ ) \Rightarrow \mathit{false}$
}
22 getfield ListArray.list

{
$\mathrm{st}(\mathrm{cntr}) = \texttt{null} \Rightarrow \mathit{false}$
$\mathrm{st}(\mathrm{cntr}) \neq \texttt{null} \wedge$
$\mathrm{st}(\mathrm{cntr}-1) < \mathit{len}(\mathrm{st}(\mathrm{cntr})) \Rightarrow$
$\mathrm{reg}_0 = \texttt{null} \Rightarrow \mathit{false} \wedge$
$\mathrm{reg}_0 \neq \texttt{null} \Rightarrow$
$\mathit{ListArray.list}(\mathrm{reg}_0) \neq \texttt{null} \wedge$
$\mathit{len}(\mathit{test.ListArray.list}(\mathrm{reg}_0)) > \mathrm{reg}_3 \wedge$
$\mathrm{reg}_3 \geq 0 \wedge$
$\mathrm{reg}_1 \neq \mathrm{ListArray.list}(\mathrm{reg}_0[\mathrm{reg}_3]) \Rightarrow$
$1 + \mathrm{reg}_3 \leq \mathrm{len}(\mathrm{test.ListArray.list}(\mathrm{reg}_0)) \wedge$
$1 + \mathrm{reg}_3 \geq 0 \wedge$
$\forall \mathit{var}(0).\ 0 \leq \mathit{var}(0) \wedge \mathit{var}(0) < 1 + \mathrm{reg}_3 \Rightarrow$
$\mathit{ListArray.list}(\mathrm{reg}_0)[\mathrm{var}(0)] \neq \mathrm{reg}_1))) \wedge$
$\mathit{ListArray.list}(\mathrm{reg}_0) = \texttt{null}\ ) \Rightarrow \mathit{false} \wedge$
$\mathit{len}(\mathit{test.ListArray.list}(\mathrm{reg}_0)) \leq \mathrm{reg}_3) \vee \mathrm{reg}_3 < 0 \Rightarrow \mathit{false}$
}
23 arraylength

{
$\mathrm{st}(\mathrm{cntr}-1) < \mathrm{st}(\mathrm{cntr}) \Rightarrow$
$\mathrm{reg}_0 = \texttt{null} \Rightarrow \mathit{false} \wedge$
$\mathrm{reg}_0 \neq \texttt{null} \Rightarrow$
$\mathit{ListArray.list}(\mathrm{reg}_0) \neq \texttt{null} \wedge$
$\mathit{len}(\mathit{test.ListArray.list}(\mathrm{reg}_0)) > \mathrm{reg}_3 \wedge$
$\mathrm{reg}_3 \geq 0 \wedge$
$\mathrm{reg}_1 \neq \mathrm{ListArray.list}(\mathrm{reg}_0[\mathrm{reg}_3]) \Rightarrow$
$1 + \mathrm{reg}_3 \leq \mathrm{len}(\mathrm{test.ListArray.list}(\mathrm{reg}_0)) \wedge$
$1 + \mathrm{reg}_3 \geq 0 \wedge$
$\forall \mathit{var}(0).\ 0 \leq \mathit{var}(0) \wedge \mathit{var}(0) < 1 + \mathrm{reg}_3 \Rightarrow$
$\mathit{ListArray.list}(\mathrm{reg}_0)[\mathrm{var}(0)] \neq \mathrm{reg}_1))) \wedge$
$\mathit{ListArray.list}(\mathrm{reg}_0) = \texttt{null}\ ) \Rightarrow \mathit{false} \wedge$
$\mathit{len}(\mathit{ListArray.list}(\mathrm{reg}_0)) \leq \mathrm{reg}_3) \vee \mathrm{reg}_3 < 0 \Rightarrow \mathit{false}$
}
24 if_icmplt6

4.4 Correctness

The correctness criteria for the wp calculus requires that if the function is applied to a bytecode program P and postcondition ψ^{post} , predicate calculated by the calculus is provable then if the postcondition upon which

5 Verification Condition Genarator for Java Bytecode Programs

6 Equivalence between Java Source and Bytecode Proof Obligations

7 Fitting a verification condition generator on a small device

8 Applications

9 Conclusion

References

[1] AV, Sethi R, and Ullman JD. *Compilers-Principles, Techniques and Tools*. Addison-Wesley: Reading, 1986.

[2] Gilles Barthe, Guillaume Dufay, Line Jakubiec, and Simao Melo de Sousa. A formal correspondence between offensive and defensive javacard virtual machines. In *VMCAI*, pages 32–45, 2002.

[3] Gilles Barthe, Guillaume Dufay, Line Jakubiec, Bernard Serpette, and Simão Melo de Sousa. A formal executable semantics of the JavaCard platform. *Lecture Notes in Computer Science*, 2028:302+, 2001.

[4] B.Meyer. *Object-Oriented Software Construction*. Prentice Hall, second revised edition edition, 1997.

[5] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G.T. Leavens, K.R.M. Leino, and E. Poll. An overview of JML tools and applications. In T. Arts and W. Fokkink, editors, *Formal Methods for Industrial Critical Systems (FMICS 2003)*, volume 80 of *ENTCS*. Elsevier, 2003.

[6] L. Burdy, A. Requet, and J.-L. Lanet. Java applet correctness: A developer-oriented approach. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods: International Symposium of Formal Methods Europe*, volume 2805 of *LNCS*, pages 422–439, 2003.

[7] Yoonsik Cheon and Gary T. Leavens. A runtime assertion checker for the Java modeling language. In *Software Engineering Research and Practice (SERP'02)*, CSREA Press, pages 322–328, June 2002.

[8] Stephen N. Freund and John C. Mitchell. A formal framework for the java bytecode language and verifier. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 147–166, New York, NY, USA, 1999. ACM Press.

[9] G.T.Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, and Joseph Kiniry. *JML Reference Manual*. technical report.

[10] Gerwin Klein and Tobias Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. Technical Report 0400001T.1, National ICT Australia, Sydney, March 2004.

[11] R.K. Leino. escjava. `http://secure.ucd.ie/products/opensource/ESCJava2/docs.html`.

[12] Tim Lindholm and Frank Yellin. Java virtual machine specification. Technical report, Java Software, Sun Microsystems, Inc., 2004.

[13] Cornelia Pusch. Proving the soundness of a java bytecode verifier in isabelle/hol, 1998.

[14] Zhenyu Qian. A formal specification of java virtual machine instructions for objects, methods and subrountines. In *Formal Syntax and Semantics of Java*, pages 271–312, 1999.