Bytecode Verification and its Applications

September 21, 2006

Contents

Chapter 1

Introduction

A natural question is to ask what are the motivations behind building a bytecode verification condition generator (vcGen for short) while a large list of tools for source code verification exists. We will try in the following to give an answer to this question. Let us start with a brief description of the context of the present thesis.

1.1 Context

Establishing trust in software components that originate from untrusted or unknown producers is an important issue for trusted personal devices (TPDs for short) such as smart cards, mobile phones, bank cards. TPDs commonly rely on execution platforms such as the Java Virtual Machine(JVM) and the Common Language Runtime. Such platforms are considered appropriate for TPDs since they allow applications to be developed in a high-level language without committing to any specific hardware and since they feature security mechanisms that guarantee the innocuousness of downloaded applications.

Depending on what exactly we want to guarantee about the security of an executable software component different techniques exist:

verification over the source code The field of source verification is wildly studied and several tools exists for dealing with source verification. We can start the listing with the Eiffel programming system upto all the verification tools tailored to Java: esc/java [46], the Loop tool [41], Krakatoa [50], Jack [20]. However, using source verification techniques for guaranteeing properties on the interpreted code requires (1) that the source code accompanies the bytecode (2) the code receiver trust the compiler.

type based verification techniques This approach does not need the presence of the source code neither require to trust the compiler as the checks can be done directly on bytecode. In particular, a typical example is the Java bytecode verifier (see [47]) which is part of the JVM. The bytecode

verifier performs static analysis over the bytecode yet, it can only guarantee that the code is well typed and well structured or in other words that the bytecode does not corrupt the performance of the virtual machine.

dynamic checks This approach consists in performing checks dynamically on execution time. However, the performance is compromiseds especially in the case of devices with limitted resources. For example, the Java security architecture ensures that applications will not perform illegal memory accesses through stack inspection, which performs access control during execution

proof carrying code The Proof Carrying Code paradigm (PCC) and the certifying compiler [56] are another alternative. In this architecture, untrusted code is accompanied by a proof for its safety w.r.t. to some safety property and the code receiver has to generate the verification conditions and type check the proof against them. The proof is generated automatically by the certifying compiler for properties like well typedness or safe memory access. As the certifying compiler is designed to be completely automatic, it will not be able to deal with rich functional or security properties.

1.2 Motivations

The present thesis addresses the issue of bytecode verification and aims at the following features:

no trust in the compiler required Our objective will be to build a verification framework which does not make assumptions on the compiler. In the case of critical software the client will not be willing to trust the compiler. In particular, he would like to get a direct evidence that the executable software component he receives will behave in the expected way

Java bytecode Java is widely used in software industry mainly because of its platform independence and its secure mechanisms. This in particular is true for the TPD industry where guarantees of software quality and security are important. From a practical point of view, we consider that Java is a good target for a verification framework.

verify complex client policies Criteria for software quality may potentially be complex. For instance, in a mobile code scenario where a client receives an untrusted implementation of an interface and he would like to get a guarantee that the implementation respects the functional specification of the interface. Note that such kind of checks may be not easily verified with type checking.

1.3 Novel approach

In order to respect the above requirements we propose a framework with the following components:

Verification condition generator for Java bytecode As we stated above, we target to build a framework where no assumptions for the compiler are made. Thus, we propose a verification condition generator (VcGen for short) for Java bytecode which is completely independent from the source code. The verification condition generator a large subset of Java and deals with arithmetic operations, object creation and manipulation, method invokations, exception throwing and handling, stack manipulation etc.

We have an implementation which is integrated in Jack (short for Java Applect Correctness Kit) [20] which is a plugin for the eclipse ide [1]

Bytecode Modeling Language A specification language which is tailored to bytecode. Let us see what advocates the need of a low level specification language. Traditionally, specification languages were tailored for high level languages. Source specification allows to express complex functional or security properties about programs. Thus, they are successfully 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 or 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, verifying a bytecode program against certain functional property needs a formalism into which the property will be encoded.

It is in this perspective, that we propose a bytecode specification language which we call BML (short for Bytecode Modeling Language). BML is the bytecode encoding of an important subset of JML (short for Java Modeling Language). The latter is a rich specification language tailored to Java source programs and allows to specify rich functional properties over Java source programs.

BML supports the most important features of JML like method pre and postconditions, intra method specification as for instance loop invariants, class invariants, special keywords. Thus, we can express functional and security 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. To our knowledge BML does not have predecessors that are tailored to Java bytecode.

[1] http://www.eclipse.org/

Compiler from source to bytecode annotations One would ask why we need a bridge between source and bytecode specification. Or put it differently, why not writing or inferring specification directly on bytecode.

For properties like well typedness specification can be inferred automatically, but in the general case this problem is not decidable. Thus, for more sophisticated policies, an automatic inference will not work. A compiler from JML to BML makes the Java bytecode benefit from a compiler from JML towards BML.

Equivalence between source and bytecode proof obligations Such an equivalence is useful when programs and requirements over them are complex. In this case, an interactive verification procedure over the source code could be helpful. (e.g. proving the verification conditions in an interactive theorem prover). An application of such an equivalence is the First, interactive procedure is suitable where automatic decision procedures will not cope with difficult theorems which is potentially the case for sophisticated security policies or functional requirements. Second, using verification on source code is useful as program bugs and errors can be easily identified and corrected. Because of the relative equivalence between source and bytecode proof obligations, once the verification conditions over the source and the program requirements expressed as specifications are proved the bytecode and the certificate (the proof of the verification conditions) produced over the source can be shipped to the client.

Such a framework may be used for different purposes. We illustrate this by two examples - verification of constraint memory consumption policies and optimization of Java to native compiler.

1.4 Applications

Chapter 2

Java bytecode language and its operational semantics

The purpose of this chapter is to introduce a bytecode language and its operational semantics. The language will be used all along the thesis and more particularly will be used later in Chapter 5 for the definition of the verification condition generator. The operational semantics of the language will be used for establishing its correctness of the verification procedure. As our verification procedure is tailored to Java bytecode the bytecode language introduced hereafter is close to the Java Virtual Machine language [49](JVM for short).

In particular, **the features supported** by our bytecode language are

arithmetic operations like multiplication, division, addition and substruction.

stack manipulation Similarly to the JVM our abstract machine is stack based, i.e. instructions get their arguments from the operand stack and push their result on the operand stack.

method invokation In the following, we consider only non void methods. We restrict our modelization for the sake of simplicity without losing any specific feature of the Java language.

object manipulation and creation Java supports all the facilities for object creation and manipulation. This is an important aspect of the Java language as it is completely object oriented. Thus, we support field access and update as well as object creation. Note that we also support arrays with the corresponding facilities for creation, access and update.

exception throwing and handling Our bytecode language supports runtime and programmatic exceptions as the JVM does. An example for a situation where a runtime exception is thrown is a null object dereference. Programmatic exceptions are forced by the programmer with a special instruction in the program text.

classes and class inheritance Like in the JVM language, our bytecode language supports a tree class hierarchy in which every class has a super class except the class `Object` which is the root of the class hiearchy.

integer type The unique basic type which is supported is the integer type. This is not so unrealistic as the JVM treats the other integral types e.g. byte and short like the integer type. JVM actually supports only few instructions for dealing in a special way with the array of byte and short.

Our bytecode language omits some of the features of Java. Let us see which ones exactly and why.

The features not supported by our bytecode language are

void methods Note that the current formalization can be extended to void methods without major difficulties.

static fields and methods Static data is shared between all the instances of the class where it is declared. We can extend our formalization to deal with static fields and methods, however it would have made the presentation heavier without gaining new feature from the JVM bytecode language.

static initialization This part of the JVM is discarded as its formal understanding is difficult and complex. Static initialization is a good candidate for a future work.

subroutines Subroutines in Java is a piece of code inside a method which must be executed after another, no matter how the first terminate excecution. They correspond to the construction `try{ } finally { }` in the Java language. Subroutines are a controversory point in the JVM because on one hand they complicate a lot the bytecode verification algorithm in Java and second, slow the JVM execution because of the way they are implemented. While the standard compilation of `try{ } finally { }` is with special instructions, the most recent Java compilers inline the subroutine code which results more efficient for the bytecode verifier as well as for the code execution. In the implementation of our verification calculus we also inline subroutines and in this way we omit the special bytecode constructs for subroutines in our bytecode language.

errors The exception mechanism in Java is provided with two kind of exceptions: exceptions from which a reasonable application may recover. i.e. handle the exception and exceptions which cannot be recovered. This last group of exceptions in Java are called JVM errors. JVM errors are thrown typically when the proper functioning of the JVM cannot continue. The cause might be an error on loading, linking, initialization time of a class or on execution time because of defficiency in the JVM resource, e.g. stack, memory overflow. For instance, during the resolution of a field name may terminate on `NoSuchFieldError` if such a field is not found. Another example is the `MemoryOverflowError` thrown when no more memory is

available. Note that such errors is not always clear how to express in our program logic presented later in the thesis. This is because, the logic is more related with the functional properties of the program while the JVM errors are related more to the physical state of the JVM. Thus, we omit here this aspect of the JVM.

interface types These are reference types whose methods are not implemented and whose variables are constants. Such interface types are then implemented by classes and allow that a class get more than one behavior. A class may implement several interfaces. The class must give an implementation for every method declared in any interface that it implements. If a class implements an interface then every object which has as type the class is also of the interface type. Interfaces are the cause of problems in the bytecode verifier as the type hierarchy is no more a lattice in the presence of interface types and thus, the least common super type of two types is not unique. However, in the current thesis we do not deal with bytecode verification but we will be interested in the program functional behaviour. For instance, if a method overrides a method from the super class or implements a method from an interface, our objective will be to establish that the method respects the specification of the method it overrides or implements. In this sense, super classes or interfaces are treated similarly in our verification tool.

Moreover, considering interfaces would have complicated the current formalization without gaining more new features of Java. For instance, in the presence of interfaces, we should have extended the subtyping relation.

arithmetic overflow The arithmetic in Java is bounded. This means that if the result of an arithmetic operation exceeds (is below) the largest integer (the smallest) the operation will result in overflow. The arithmetic proposed here is infinite. We could have designed the arithmetic operations such that they take into account the arithmetic overflow. However, we consider that for the purposes of the present thesis this is not necessary and will complicate the presentation without bringing any particular feature of the bytecode.

64 bit arithmetic We do not consider long values as their treatment is similar to the integer arithmetic. However, it is true that the formalization and manipulation of the long type can be more complicated as long values are stored in two adjacent registers but it is feasible to extend the current formalization to deal with long values.

floating point arithmetic We omit this data in our bytecode language for the following reasons. There is no support for floating point data by automated tools. For instance, the automatic theorem prover Simplify which interfaces our verification tool lacks support for floating point data, see [45]. Although larger and more complicated than integral data, formalization of floating point arithmetic is possible. For example, the specification

of IEEE [25] for floating point arithmetic as well as a proof for its consistency is done in the interactive theorem prover Coq. However, including floating point data would not bring any interesting part of Java but would rather turn more complicated and less understandable the formalizations in the current document.

Now that we have seen the general outlines of our language, in the rest of this chapter we shall proceed with a more detailed description.

The rest of this chapter is organized as follows: 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 introduces the notion of state configuration, subsection 2.5.1 gives the modelisation of the memory heap, subsection 2.7 is a discussion about our choice for operational semantics, 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 JVM semantics as a reply to the holes and ambiguities encountered in the specification of the Java bytecode verifier. Thus, most of the existing formalizations are used for reasoning over the Java bytecode well-typedness.

We can start with the work of Stata and Abadi [68] in which they propose a semantics and typying rules for checking the correct behavior of subroutines in the presence of polymorphism. In [30] N.Freund and J.Mitchell extend the aforementioned work to a language which supports all the Java features e.g. object manipulation and instance initialization, exception handling, arrays. In [60] Qian gives 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 is used for the formal specification of the language and the bytecode verifier. Based on the work of Qian, in [59] 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 [42], 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 as 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 (the JVM version tailored to smart cards) simplifies the formalization of the system and thus, has attracted particularly the scientific interest. CertiCartes [14, 13] 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. Siveroni proposes a formalization of the JCVM in [67] in terms of a small step operational semantics.

2.2 Notation

Here we introduce several notations used in the rest of this chapter. 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 $A_1 \ \ldots \ A_n$ respectively and maps them to elements of type B we note the function signature with $f : A_1 * ... * A_n \rightarrow B$. The set of elements which represent the domain of the function f is given by the function $\mathsf{Dom}(f)$ and the elements in its range are given by $\mathsf{Range}(f)$.

Function updates of function f with n arguments is denoted with $f[\oplus x_1 \ldots x_n \rightarrow y]$ and the definition of such function is :

$$f[\oplus x_1 \ldots x_n \rightarrow y](z_1 \ldots z_n) = \begin{cases} y & if\ x_1 = z_1 \wedge ... \wedge x_n = z_n \\ f(z_1 \ldots z_n) & else \end{cases}$$

The type $list$ is used to represent a sequence of elements. The empty list is denoted with [ ]. If it is true that the element e is in the list l, we use the notation $e \in l$. The function :: receives two arguments an element e and a list l and returns a new list $e::l$ whose head and tail are respectively e and l. The number of elements in a list l is denoted with $l.length$. The i-th element in a list l is denoted with $l[i]$. Note that the indexing in a list l starts at 0, thus the last index in l being $l.length - 1$.

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, classes are encoded with the data structure **Class**, fields with **Field** and methods are encoded with **Method** data structure. 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). All classes, except the special class `Object` , have a unique direct super class. Formally, a class of our bytecode language has the following structure:

$$\mathbf{Class} = \left\{ \begin{array}{ll} \mathsf{fields} & : \mathit{list}\ \mathbf{Field} \\ \mathsf{methods} & : \mathit{list}\ \mathbf{Method} \\ \mathsf{className} & : \mathbf{ClassName} \\ \mathsf{superClass} & : \mathbf{Class} \cup \{\bot\} \end{array} \right\}$$

A field object is a tuple that contains the unique field id (Name) and a field type (Type) and the class where it is declared (declaredIn):

$$\mathbf{Field} = \left\{ \begin{array}{ll} \mathsf{Name} & : \mathbf{FieldName}; \\ \mathsf{Type} & : \mathit{JType}; \\ \mathsf{declaredIn} & : \mathbf{Class} \cup \{\bot\} \end{array} \right\}$$

From the above definition, we can notice that the field declaredIn may have a value $\bot$. This is because we model the length of a reference pointing to an array object as an element from the set **Field** . Because the length of an array is not declared in any class, we assign to its attribute declaredIn the value $\bot$. The special field which stands for the array length (the name of the object and its field Name have the same name) is the following:

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

There are other possible approaches for modeling the array length. For instance, the array length can be part of the array reference. We consider that both of the choices are equivalent. However, the current formalization follows closely our implementation of the verification condition generator which encodes in this way array length which is necessary if we want to do a proof of correctness of the implementation.

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), the exception handler table (excHndlS) and the list of exceptions (exceptions) that the method may throw

$$\mathbf{Method} = \left\{ \begin{array}{ll} \mathsf{Name} & : \mathbf{MethodName} \\ \mathsf{retType} & : \mathit{JType} \\ \mathsf{args} & : \mathit{list}\ (\mathit{name} * \mathit{JType}) \\ \mathsf{nArgs} & : \mathit{nat} \\ \mathsf{body} & : \mathit{list}\ \ \mathrm{I} \\ \mathsf{excHndlS} & : \mathit{list}\ \mathbf{ExcHandler} \\ \mathsf{exceptions} & : \mathit{list}\ \mathbf{Class}_{exc} \end{array} \right\}$$

We assume that for every method `m` the entrypoint is the first instruction in the list of instructions of which the method body consists, i.e. `m`.entryPnt = `m`.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.

$$\mathbf{ExcHandler} = \left\{ \begin{array}{ll} \mathsf{startPc} & : nat \\ \mathsf{endPc} & : nat \\ \mathsf{handlerPc} & : nat \\ \mathsf{exc} & : \mathbf{Class}_{exc} \end{array} \right\}$$

We require that startPc, endPc and handlerPc fields in any exception handler attribute m.excHndlS for any method m are valid indexes in the list of instructions of the method body m.body:

$$\begin{array}{l} \forall \mathtt{m} : \mathbf{Method}, \\ \forall i : nat, 0 \leq i < \mathtt{m}.\mathsf{excHndlS}.length, \\ \quad 0 \leq \mathtt{m}.\mathsf{excHndlS}[i].\mathsf{endPc} < \mathtt{m}.\mathsf{body}.length \wedge \\ \quad 0 \leq \mathtt{m}.\mathsf{excHndlS}[i].\mathsf{startPc} < \mathtt{m}.\mathsf{body}.length \wedge \\ \quad 0 \leq \mathtt{m}.\mathsf{excHndlS}[i].\mathsf{handlerPc} < \mathtt{m}.\mathsf{body}.length \end{array}$$

2.4 Program types and values

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 (*RefTypeCl*) and array reference types (*RefTypeArr*). As we said in the beginning of this chapter, the language does not support interface types.

$$\begin{array}{ll} \mathit{JType} & ::= \mathbf{int} \mid \mathit{RefType} \\ \mathit{RefType} & ::= \mathit{RefTypeCl} \mid \mathit{RefTypeArr} \\ \mathit{RefTypeCl} & ::= \mathbf{Class} \\ \mathit{RefTypeArr} & ::= \mathit{JType}[] \end{array}$$

Our language supports two kinds of values : values of the basic type **int** and reference values *RefVal*. *RefVal* may be references to class objects, references to array objects or the special null value which denotes the reference pointing nowhere. The set of references of class objects is denoted with *RefValCl*, the set of references to array objects is represented with *RefValArr*and the null reference value is denoted with **null**. The following definition gives the formal grammar for values:

$$\begin{array}{ll} \mathit{Values} & ::= i \mid \mathit{RefVal} \\ \mathit{RefVal} & ::= \mathit{RefValCl} \mid \mathit{RefValArr} \mid \mathbf{null} \end{array}$$

Every type has an associated default value which can be accessed via the function defVal. Particularly, for reference types (*RefType*) the default value is **null** and the default value of **int** type is 0. Thus, the definition of the function defVal is as follows:

$$\mathsf{defVal} : \mathit{JType} \rightarrow \mathit{Values}$$

$$\mathsf{defVal}(\mathtt{T}) = \begin{cases} \textbf{null} & \mathtt{T} \in \mathit{RefType} \\ 0 & \mathtt{T} = \textbf{int} \end{cases}$$

We define also a subtyping relation as follows:

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

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

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

Note that the subtyping relation that we use here is a subset of the Java subtyping relation. However, it is a subset of the Java's subtyping relation which includes as well interface and abstract class types.

2.5 State configuration

In this section, we introduce the notion of program state. A state configuration Smodels the program state in particular execution program point by specifying what is the memory heap in the state, 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. Note that, as we stated before our semantics ignores the method call stack and so, state configurations also omit the call frames stack.

We define two kinds of state configurations:

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

The set S^{interm} consists of method intermediate state configurations, which stand for an *intermediate 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 $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \in S^{interm}$ has the following elements:

- the function H: $\mathsf{HeapType}$ which stands for the heap in the state configuration
- Cntr is a variable that contains a natural number which stands for the number of elements in the operand stack.
- Stis a partial function from natural numbers to values which stands for the operand stack.
- Reg is a partial function from natural numbers to values which stands for the array of local variables of a method. Thus, for an index `i` it returns the value $\mathbf{reg}(i)$ which is stored at that index of the array of local variables

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

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

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

- $< \mathrm{H}, \mathrm{Res} >^{norm} \in S^{norm}$ which describes a *normal final state*, i.e. the method execution terminates normally. The normal termination configuration has the following components :
  - the function H: HeapType which reflects what is the heap state after the method terminated
  - Res stands for the return value of the method
- $< \mathrm{H}, \mathrm{Exc} >^{exc} \in S^{exc}$ 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

When an element of a state configuration $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ is updated we use the notation:

$$S[E \backslash V], \ E \in \{\mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc}\}$$

We will denote with $< \mathrm{H}, \mathit{Final} >^{final}$ for any configuration which belongs to the set S^{final}. Later on in this chapter, we define in terms of state configuration transition relation the operational semantics of our bytecode programming language. In the following, we focus in more detail on the heap modelization and the operand stack.

2.5.1 Modeling the object heap

An important issue for the modelization of an object oriented programming language and its operational semantics is the memory heap. 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. We do not take into account garbage collection and thus, we assume that heap objects has an infinite memory space.

In our modelization, a heap consists of the following components:

- a component named Fld which is a partial function that maps field structures (of type **Field** introduced in subsection 2.3) into partial functions from references (*RefType*) into values (*Values*).
- a component Arr which maps the components of arrays into their values
- a component Loc which stands for the list of references that the heap has allocated
- a component TypeOf is a partial function which maps references to their dynamic type

Formally, the data type HeapType has the following structure:

$$\forall \mathrm{H} : \mathsf{HeapType},$$
$$\mathrm{H} = \left\{ \begin{array}{ll} \mathsf{Fld} & : \mathbf{Field} \rightharpoonup (\mathit{RefVal} \rightharpoonup \mathit{Values}) \\ \mathsf{Arr} & : \mathit{RefValArr} * \mathit{nat} \rightharpoonup \mathit{Values} \\ \mathsf{Loc} & : \mathit{list}\ \mathit{RefVal} \\ \mathsf{TypeOf} & : \mathit{RefVal} \rightharpoonup \mathit{RefType} \end{array} \right\}$$

Another possibility is to model the heap as partial function from locations to objects where objects contain a function from fields to values. Both formalizations are equivalent, still we have chosen this model as it follows closely the verification condition generator implementation.

In the following, we are interested only in heap objects H for which the components H.Fld and H.Arr are functions defined only for references from the proper type, i.e. well-typed and which are in the list of references of the heap H.Loc, i.e. well-defined:

$$\begin{array}{ll} \forall f : \mathbf{Field}, \forall \mathtt{ref} \in \mathit{RefVal}, & \mathtt{ref} \in \mathsf{Dom}(\mathrm{H}.\mathsf{Fld}(f)) \Rightarrow \\ & \mathtt{ref} \in \mathrm{H}.\mathsf{Loc} \wedge \\ & \mathsf{subtype}(\mathrm{H}.\mathsf{TypeOf}\ (\mathtt{ref}), f.\mathsf{declaredIn}) \\ \wedge & \\ \forall \mathtt{ref} \in \mathit{RefValArr}, & (\mathtt{ref}, i) \in \mathsf{Dom}(\mathrm{H}.\mathsf{Arr}) \Rightarrow \\ & \mathtt{ref} \in \mathrm{H}.\mathsf{Loc} \wedge \\ & 0 \leq i < \mathrm{H}.\mathsf{Fld}(\mathrm{arrLength})(\mathtt{ref}) \end{array}$$

If a new object of class C is created in the memory, a fresh reference value `ref` different from **null** 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. The function which for a heap H and a class type C returns the same heap but with a fresh reference of type C has the following name and signature:

$$\mathrm{newRef} : \mathrm{H} \rightarrow \mathit{RefTypeCl} \ \rightarrow \mathrm{H} * \mathit{RefValCl}$$

The formalization of the resulting heap and the new reference is the following:

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

$$\begin{array}{l}
\texttt{ref} \neq \textbf{null} \wedge \\
\texttt{ref} \notin \text{H.Loc} \wedge \\
\text{H}'.\text{Loc} = \texttt{ref}::\text{H.Loc} \wedge \\
\text{H}'.\text{TypeOf} \ := \text{H.TypeOf} \ [\oplus \texttt{ref} \rightarrow C] \wedge \\
\ \forall f : \textbf{Field}, \quad \mathit{instFlds}(f, C) \Rightarrow \\
\qquad\qquad\qquad \text{H}'.\text{Fld} := \text{H}'.\text{Fld}[\oplus f \rightarrow f[\oplus \texttt{ref} \rightarrow \text{defVal}(f.\text{Type})]] \wedge
\end{array}$$

In the above definition, we use the function *instFlds*, which for a given field f and C returns true if f is an instance field of C:

$$\mathit{instFlds} : \textbf{Field} \rightarrow \textbf{Class} \rightarrow \mathit{bool}$$

$$\mathit{instFlds}(f, C) = \begin{cases} \mathit{true} & f.\text{declaredIn} = C \\ \mathit{false} & C = \texttt{Object} \wedge f.\text{declaredIn} \neq \texttt{Object} \\ \mathit{instFlds}(f, C.\text{superClass}) & \mathit{else} \end{cases}$$

Identically, when allocating a new object of array type whose elements are of type `T` and length l, we obtain a new heap object $\text{newArrRef}(\text{H}, \texttt{T}[\,], l)$ which is defined similarly to the previous case:

$$\text{newArrRef} : \text{H} \rightarrow \mathit{RefTypeArr} \rightarrow \text{H} * \mathit{refArr}$$

$$\begin{array}{l}
\text{newArrRef}(\text{H}, \texttt{T}[\,], l) \ = (\text{H}', \texttt{ref}) \iff^{def} \\
\\
\texttt{ref} \neq \textbf{null} \wedge \\
\texttt{ref} \notin \text{H.Loc} \wedge \\
\text{H}'.\text{Loc} = \texttt{ref}::\text{H.Loc} \wedge \\
\text{H}'.\text{TypeOf} \ := \text{H.TypeOf} \ [\oplus \texttt{ref} \rightarrow \texttt{T}[\,]] \wedge \\
\text{H}'.\text{Fld} := \text{H}'.\text{Fld}[\oplus \text{arrLength} \rightarrow \text{arrLength}[\oplus \texttt{ref} \rightarrow l]] \wedge \\
\forall i, 0 \leq i < l \Rightarrow \text{H}'.\text{Arr} := \text{H}'.\text{Arr}[\oplus(\texttt{ref}, i) \rightarrow \text{defVal}(\texttt{T})]
\end{array}$$

In the following, we adopt few more naming conventions which do not create any ambiguity. Getting the function corresponding to a field f in a heap H : $\text{H.Fld}(f)$ is replaced with $\text{H}(f)$ for the sake of simplicity.

The same abreviation is done for access of an element in an array object referenced by the reference `ref`at index i in the heap H. Thus, the usual denotation: $\text{H.Arr}(\texttt{ref}, i)$ becomes $\text{H}(\texttt{ref}, i)$.

Whenever the field f for the object pointed by reference `ref` is updated with the value *val*, the component H.Fld is updated:

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

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

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

If in the heap H the i^{th} component in the array referenced by `ref` is updated with the new value *val*, this results in assigning a new value of the component H.Arr:

$$\mathrm{H.Arr} := \mathrm{H.Arr}[\oplus(\texttt{ref}, i) \rightarrow val]$$

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:

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

2.5.2 Registers

State configurations have an array of registers which is denoted with Reg. Registers are addressed by indexing and the index of the first local variable is zero. Thus, Reg(0) stands for the first register in the state configuration. An integer is be considered to be an index into the local variable array if and only if that integer is between zero and one less than the size of the local variable array. Registers are used to pass parameters on method invocation. On class method invocation any parameters are passed in consecutive local variables starting from register Reg(0). Reg(0) is always used to pass a reference to the object on which the instance method is being invoked (`this` in the Java programming language). Any parameters are subsequently passed in consecutive local variables starting from local variable 1.

2.5.3 The operand stack

Like the JVM language, our bytecode language is stack based. This means that every method is supplied with a Last In First Out stack which is used for the method execution to store intermediate results. The method stack is modeled by the partial function St and the variable Cntr keeps track of the number of the elements in the operand stack. St is defined for any integer `ind` smaller than the operand stack counter Cntr and returns the value St(`ind`) stored in the operand stack at `ind` positions of the bottom of the stack. When a method starts execution its operand stack is empty and we denote the empty stack with [ ]. Like in the JVM our language supports instructions to load values stored in registers or object fields and viceversa. There are also instructions that take their arguments from the operand stack St, operate on them and push the result on the operand stack. The operand stack is also used to prepare parameters to be passed to methods and to receive method results.

2.5.4 Program counter

The last component of an intermediate state configuration is the program counter Pc. It contains the number of the instruction in the array of instructions of the current method which must be executed in the state.

2.6 Throwing and handling exceptions

As the JVM specification states *exception are thrown if a program violates the semantic constraints of the Java programming language, the Java virtual machine signals this error to the program as an exception. An example of such a violation is an attempt to index outside the bounds of an array. The Java programming language specifies that an exception will be thrown when semantic constraints are violated and will cause a nonlocal transfer of control from the point where the exception occurred to a point that can be specified by the programmer. An exception is said to be thrown from the point where it occurred and is said to be caught at the point to which control is transferred. A method invocation that completes because an exception causes transfer of control to a point outside the method is said to complete abruptly. Programs can also throw exceptions explicitly, using throw statements ...*

Our language supports an exception handling mechanism similar to the JVM one. More particularly, it supports Runtime exceptions:

- `NullPntrExc` thrown if a null pointer is dereferenced
- `NegArrSizeExc` thrown if there is an attempt to creat an array with a negative size
- `ArrIndBndExc` thrown if an array is accessed out of its bounds
- `ArithExc` thrown if a division by zero is done
- `CastExc` thrown if an object reference is cast to to an incompatible type
- `ArrStoreExc` thrown if an object is tried to be stored in an array and the object is of incompatible type with type of the array elements

The language also supports programming exceptions. Those exceptions are forced by the programmer, by a special bytecode intruction as we shall see later in the coming section.

The modelization of the exception handling mechanism involves several auxiliary functions. The function *getStateOnExcRT* deals with bytecode instructions that may throw runtime exceptions. This function applies only to instructions which may throw a `RuntimeExc` exception but which are not a method invokation neither the special instruction by which the program can throw explicitely an exception. The function returns the state configuration after the current instruction during the execution of `m` throws a runtime 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 will proceed and thus, the state 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. Note also that the heap is changed as a new instance of the corresponding exceptional type is created:

$$getStateOnExcRT : S^{interm} * ExcType * \mathbf{ExcHandler}[] \rightarrow S^{interm} \cup S^{exc}$$

$$getStateOnExcRT(< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >, \mathtt{E}, \mathsf{excH}[]) =$$
$$\begin{cases} < \mathrm{H}', 0, \mathrm{St}[\oplus 0 \rightarrow \mathtt{ref}], \mathrm{Reg}, \mathsf{handlerPc} > & \text{if } findExcHandler(\mathtt{E},\mathrm{Pc},\mathsf{excH}[]) = \mathsf{handlerPc} \\ < \mathrm{H}', \mathtt{ref} >^{exc} & \text{if } findExcHandler(\mathtt{E},\mathrm{Pc},\mathsf{excH}[]) = \bot \end{cases}$$

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

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 exists such a handler the function returns the index of the instruction at which the exception handler starts, otherwise it returns $\bot$:

$$findExcHandler : ExcType * nat * \mathbf{ExcHandler}[] \rightarrow \{nat \cup \bot\}$$

$$findExcHandler(\ \mathtt{E}, \mathrm{Pc}, \mathsf{excH}[]) =$$
$$\begin{cases} \mathsf{excH}[m].\mathsf{handlerPc} & if\ hExc \neq emptySet \\ & where\ m = min(hExc) \\ \bot & else \end{cases}$$

where
$$hExc = \{k \mid \begin{array}{l} \mathsf{excH}[k] = (\mathsf{startPc}, \mathsf{endPc}, \mathsf{handlerPc}, \mathtt{E}') \wedge \\ \mathsf{startPc} \leq \mathrm{Pc} < \mathsf{endPc} \wedge \\ \mathsf{subtype}(\mathtt{E}, \mathtt{E}') \end{array} \}$$

2.7 Design choices for the operational semantics

Before proceeding with the motivations for the choice of the operational semantics, we shall first look at a brief description of the semantics of the Java Virtual Machine (JVM).

JVM is stack based and when 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. Thus, an operational semantics which follows closely the JVM would model the method frame stack and use a small step operational semantics.

However, the purpose of the operational semantics presented in this chapter is to give a model w.r.t.which a proof of correctness of our verification calculus will be done. Because the latter is modular and assumes program termination, i.e. the verification calculus assumes the correctness and the termination of the rest of the methods, we do not need a model for reasoning about the termination or the correctness of invoked methods. A big step operational semantics which is silent about the method frame stack provides a suitable level of abstraction.

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. 2.1, we give the list of instructions that constitute our bytecode language.

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

We define the operational semantics of a single Java instruction as a relation between its initial and final state configurations as follows.

Definition 2.8.1 (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*

$$\mathtt{m} \vdash \mathrm{I} :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow < \mathrm{H}', \mathrm{Cntr}', \mathrm{St}', \mathrm{Reg}', \mathrm{Pc}' >$$

We also define the transitive closure of a single execution step with the next definition.

Definition 2.8.2 (Transitive closure of a method state transition relation) *If the method* m *starts execution in a state* $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *at the entry*

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

Figure 2.1: BYTECODE LANGUAGE

point instruction $\mathtt{m}.\mathsf{body}[0]$ *and there exists a transitive state transition to the state* $< \mathrm{H}', \mathrm{Cntr}', \mathrm{St}', \mathrm{Reg}', \mathrm{Pc}' >$ *we denote this with:*

$$< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow^* < \mathrm{H}', \mathrm{Cntr}', \mathrm{St}', \mathrm{Reg}', \mathrm{Pc}' >$$

The following definition characterizes the executions that terminate.

Definition 2.8.3 (Termination of method execution) *If the method* $\mathtt{m}$ *starts execution in a state* $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *at the entry point instruction* $\mathtt{m}.\mathsf{body}[0]$ *and there is a transitive state transition to* $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, k >$ *such that the instruction* $\mathtt{m}.\mathsf{body}[k]$ *is either a* return *instruction or an instruction which terminates execution with an uncaught exception and the configuration after its execution is* $< \mathrm{H}', \mathit{Final} >^{final}$ *then we denote this with:*

$$\mathtt{m} :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \Rightarrow < \mathrm{H}', \mathit{Final} >^{final}$$

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

$$\frac{\mathtt{m} \vdash \mathtt{m}.\mathsf{body}[0] :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \hookrightarrow^* < \mathrm{H}', \mathit{Final} >^{final}}{\mathtt{m} :< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} > \Rightarrow < \mathrm{H}', \mathit{Final} >^{final}}$$

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 2.5. Note that we do not model the method frame stack of the JVM which is not needed for our purposes.

Fig. 2.2 shows the rules for the instructions which cause control transfer. The first rule refers to the instruction if_cond. The condition `cond` $= \{=, \neq, \leq, <, >, \geq\}$ is applied to the stack top St (Cntr) and the element below the stack top St (Cntr -1)which must be of type **int**. 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. The rule for goto shows that the instruction transfers the control to the instruction at position n. The instruction return causes the normal termination of the execution of the current method `m`. The instruction does not affect changes on the heap H and the return result is contained in the stack top element St (Cntr).

$$\frac{\texttt{cond}(\text{St (Cntr)},\text{St (Cntr}-1\text{)})}{\text{m} \vdash \text{if\_cond } n{:}\langle \text{H,Cntr,St,Reg,Pc}\rangle \hookrightarrow \langle \text{H,Cntr}-2\text{,St,Reg,}n\rangle}$$

$$\frac{not(\ \texttt{cond}(\text{St (Cntr)},\text{St (Cntr}-1\text{)}))}{\text{m} \vdash \text{if\_cond } n{:}\langle \text{H,Cntr,St,Reg,Pc}\rangle \hookrightarrow \langle \text{H,Cntr}-2\text{,St,Reg,Pc}+1\rangle}$$

$$\frac{}{\text{m} \vdash \text{return:}\langle \text{H,Cntr,St,Reg,Pc}\rangle \hookrightarrow \langle \text{H,St (Cntr)}\rangle^{norm}}$$

Figure 2.2: Operational semantics for control transfer instructions

Fig. 2.3 shows the arithmetic instructions and and the ones for stack loading and storing. Arithmetic instruction pop the values which are on the stack top St (Cntr) and St (Cntr - 1) at the position below and apply the corresponding arithmetic operation 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). Note that our formalization does not take into consideration overflow of arithmetic instructions. Here, we assume that we can manipulate any unbounded integers. Note that there are two cases for the arithmetic instructions div and rem. This is because they may terminate on a runtime exception when the second argument is 0. From the rule for exceptional termination we can see that the exception handler table `m.excHndlS` of the current method will be searched for an exception handler protecting the current position Pc from `ArithExc` exceptions and depending whether such a handler was found or not the instruction execution will terminate exceptionally or not. The instruction load increments the stack counter Cntr and pushes

the content of the local variable Reg(i) on the stack top St (Cntr + 1). The instruction store pops the stack top element St (Cntr) and stores it into local variable Reg(i) and decrements the stack counter Cntr . The instruction iinc increments the value of the local variable Reg$i$. The instruction push pushes on the stack top the integer value i and increments the stack counter Cntr . The instruction pop pops the stack top element St(Cntr). The instruction dup duplicates the stack top element St(Cntr).

$$\frac{\begin{array}{l}\mathrm{Cntr}' = \mathrm{Cntr} - 1\\ \mathrm{St}' = \mathrm{St}[\oplus \mathrm{Cntr} - 1 \rightarrow \mathrm{St}\ (\ \mathrm{Cntr}\ )\ \mathrm{op}\ \mathrm{St}\ (\ \mathrm{Cntr} - 1\ )]\\ \mathrm{Pc}' = \mathrm{Pc} + 1\end{array}}{\mathrm{m} \vdash op{:}\langle \mathrm{H,Cntr,St,Reg,Pc}\rangle \hookrightarrow \langle \mathrm{H,Cntr}',\mathrm{St}',\mathrm{Reg,Pc}'\rangle}$$

$$\frac{\begin{array}{l}op = \{\mathrm{div}, \mathrm{rem}\}\\ \mathrm{St(Cntr)} = 0\\ \mathit{getStateOnExcRT}(<\mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc}>, \texttt{ArithExc}, \mathrm{m.excHndlS}) = S\end{array}}{\mathrm{m} \vdash op{:}\langle \mathrm{H,Cntr,St,Reg,Pc}\rangle \hookrightarrow S}$$

$$\frac{\begin{array}{l}\mathrm{Cntr}' = \mathrm{Cntr} + 1\\ \mathrm{St}' = \mathrm{St}[\oplus \mathrm{Cntr} + 1 \rightarrow \mathrm{Reg}(i)]\\ \mathrm{Pc}' = \mathrm{Pc} + 1\end{array}}{\mathrm{m} \vdash \mathrm{load}\ i{:}\langle \mathrm{H,Cntr,St,Reg,Pc}\rangle \hookrightarrow \langle \mathrm{H,Cntr}',\mathrm{St}',\mathrm{Reg,Pc}'\rangle}$$

$$\frac{\begin{array}{l}\mathrm{Cntr}' = \mathrm{Cntr} - 1\\ \mathrm{Reg}' = \mathrm{Reg}[\oplus i \rightarrow \mathrm{St}\ (\ \mathrm{Cntr}\ )]\\ \mathrm{Pc}' = \mathrm{Pc} + 1\end{array}}{\mathrm{m} \vdash \mathrm{store}\ i{:}\langle \mathrm{H,Cntr,St,Reg,Pc}\rangle \hookrightarrow \langle \mathrm{H,Cntr}',\mathrm{St,Reg}',\mathrm{Pc}'\rangle}$$

$$\frac{\begin{array}{l}\mathrm{Reg}' = \mathrm{Reg}[\oplus i \rightarrow \mathrm{Reg}(i) + 1]\\ \mathrm{Pc}' = \mathrm{Pc} + 1\end{array}}{\mathrm{m} \vdash \mathrm{iinc}\ i{:}\langle \mathrm{H,Cntr,St,Reg,Pc}\rangle \hookrightarrow \langle \mathrm{H,Cntr,St,Reg}',\mathrm{Pc}'\rangle}$$

$$\frac{\begin{array}{l}\mathrm{Cntr}' = \mathrm{Cntr} + 1\\ \mathrm{St}' = \mathrm{St}[\oplus \mathrm{Cntr} + 1 \rightarrow i]\\ \mathrm{Pc}' = \mathrm{Pc} + 1\end{array}}{\mathrm{m} \vdash \mathrm{push}\ i{:}\langle \mathrm{H,Cntr,St,Reg,Pc}\rangle \hookrightarrow \langle \mathrm{H,Cntr}',\mathrm{St}',\mathrm{Reg,Pc}'\rangle}$$

$$\frac{\begin{array}{l}\mathrm{Cntr}' = \mathrm{Cntr} - 1\\ \mathrm{Pc}' = \mathrm{Pc} + 1\end{array}}{\mathrm{m} \vdash \mathrm{pop}{:}\langle \mathrm{H,Cntr,St,Reg,Pc}\rangle \hookrightarrow \langle \mathrm{H,Cntr}',\mathrm{St,Reg,Pc}'\rangle}$$

$$\frac{\begin{array}{l}\mathrm{Cntr}' = \mathrm{Cntr} + 1\\ \mathrm{St}' = \mathrm{St}[\oplus \mathrm{Cntr} + 1 \rightarrow \mathrm{St(Cntr)}]\\ \mathrm{Pc}' = \mathrm{Pc} + 1\end{array}}{\mathrm{m} \vdash \mathrm{dup}{:}\langle \mathrm{H,Cntr,St,Reg,Pc}\rangle \hookrightarrow \langle \mathrm{H,Cntr}',\mathrm{St}',\mathrm{Reg,Pc}'\rangle}$$

Figure 2.3: OPERATIONAL SEMANTICS FOR ARITHMETIC AND LOAD STORE INSTRUCTIONS

$$\frac{\begin{array}{l}\text{St (Cntr} - 1\text{)} \neq \textbf{null}\\ \text{H}' = \text{H}[\oplus f \rightarrow f[\oplus\text{St (Cntr} - 1\text{)} \rightarrow \text{St (Cntr)}]]\\ \text{Cntr}' = \text{Cntr} - 2\\ \text{Pc}' = \text{Pc} + 1\end{array}}{\text{m}\vdash\text{putfield } f{:}\langle\text{H,Cntr,St,Reg,Pc}\rangle\hookrightarrow\langle\text{H}',\text{Cntr}',\text{St,Reg,Pc}'\rangle}$$

$$\frac{\begin{array}{l}\text{St (Cntr} - 1\text{)} = \textbf{null}\\ \mathit{getStateOnExcRT}(<\text{H, Cntr, St, Reg, Pc}>,\ \texttt{NullPntrExc}, \texttt{m.excHndlS}) = S\end{array}}{\text{m}\vdash\text{putfield } f{:}\langle\text{H,Cntr,St,Reg,Pc}\rangle\hookrightarrow S}$$

$$\frac{\begin{array}{l}\text{St (Cntr)} \neq \textbf{null}\\ \text{St}' = \text{St}[\oplus\text{Cntr} \rightarrow \text{H}(f)(\text{St (Cntr)})]\\ \text{Pc}' = \text{Pc} + 1\end{array}}{\text{m}\vdash\text{getfield } f{:}\langle\text{H,Cntr,St,Reg,Pc}\rangle\hookrightarrow\langle\text{H,Cntr,St}',\text{Reg,Pc}'\rangle}$$

$$\frac{\begin{array}{l}\text{St (Cntr)} = \textbf{null}\\ \mathit{getStateOnExcRT}(<\text{H, Cntr, St, Reg, Pc}>,\ \texttt{NullPntrExc}, \texttt{m.excHndlS}) = S\end{array}}{\text{m}\vdash\text{getfield } f{:}\langle\text{H,Cntr,St,Reg,Pc}\rangle\hookrightarrow S}$$

$$\frac{\begin{array}{l}\text{St (Cntr} - 2\text{)} \neq \textbf{null}\\ 0 \leq \text{St (Cntr} - 1\text{)} < \text{arrLength(St (Cntr} - 2\text{))}\\ \text{H}' = \text{H}[\oplus(\text{St (Cntr} - 2\text{), St (Cntr} - 1\text{)}) \rightarrow \text{St (Cntr)}]\\ \text{Cntr}' = \text{Cntr} - 3\\ \text{Pc}' = \text{Pc} + 1\end{array}}{\text{m}\vdash\text{type\_astore }{:}\langle\text{H,Cntr,St,Reg,Pc}\rangle\hookrightarrow\langle\text{H}',\text{Cntr}',\text{St,Reg,Pc}'\rangle}$$

$$\frac{\begin{array}{l}\text{St (Cntr} - 2\text{)} = \textbf{null}\\ \mathit{getStateOnExcRT}(<\text{H, Cntr, St, Reg, Pc}>,\ \texttt{NullPntrExc}, \texttt{m.excHndlS}) = S\end{array}}{\text{m}\vdash\text{type\_astore }{:}\langle\text{H,Cntr,St,Reg,Pc}\rangle\hookrightarrow S}$$

$$\frac{\begin{array}{l}\text{St (Cntr} - 2\text{)} \neq \textbf{null}\\ (\text{St (Cntr} - 1\text{)} < 0 \vee\\ \text{St (Cntr} - 1\text{)} \geq \text{arrLength(St (Cntr} - 2\text{))})\\ \mathit{getStateOnExcRT}(<\text{H, Cntr, St, Reg, Pc}>,\ \texttt{ArrIndBndExc}, \texttt{m.excHndlS}) = S\end{array}}{\text{m}\vdash\text{type\_astore }{:}\langle\text{H,Cntr,St,Reg,Pc}\rangle\hookrightarrow S}$$

$$\frac{\begin{array}{l}\text{St (Cntr} - 1\text{)} \neq \textbf{null}\\ \text{St (Cntr)} \geq 0\\ \text{St (Cntr)} < \text{arrLength(St (Cntr} - 1\text{))}\\ \text{Cntr}' = \text{Cntr} - 1\\ \text{St}' = \text{St}[\oplus\text{Cntr} - 1 \rightarrow \text{H(St (Cntr} - 1\text{)St (Cntr))}]\\ \text{Pc}' = \text{Pc} + 1\end{array}}{\text{m}\vdash\text{type\_aload }{:}\langle\text{H,Cntr,St,Reg,Pc}\rangle\hookrightarrow\langle\text{H,Cntr}',\text{St}',\text{Reg,Pc}'\rangle}$$

$$\frac{\begin{array}{l}\text{St (Cntr} - 1\text{)} = \textbf{null}\\ \mathit{getStateOnExcRT}(<\text{H, Cntr, St, Reg, Pc}>,\ \texttt{NullPntrExc}, \texttt{m.excHndlS}) = S\end{array}}{\text{m}\vdash\text{type\_aload }{:}\langle\text{H,Cntr,St,Reg,Pc}\rangle\hookrightarrow S}$$

$$\frac{\begin{array}{l}\text{St (Cntr} - 1\text{)} \neq \textbf{null}\\ (\text{St (Cntr)} < 0 \vee\\ \text{St (Cntr)} \geq \text{arrLength(St (Cntr} - 1\text{))})\\ \mathit{getStateOnExcRT}(<\text{H, Cntr, St, Reg, Pc}>,\ \texttt{ArrIndBndExc}, \texttt{m.excHndlS}) = S\end{array}}{\text{m}\vdash\text{type\_aload }{:}\langle\text{H,Cntr,St,Reg,Pc}\rangle\hookrightarrow S}$$

$$\frac{\begin{array}{l}\text{St (Cntr)} \neq \textbf{null}\\ \text{St}' = \text{St}[\oplus\text{Cntr} \rightarrow \text{H(arrLength)(St (Cntr))}]\\ \text{Pc}' = \text{Pc} + 1\end{array}}{\text{m}\vdash\text{arraylength:}\langle\text{H,Cntr,St,Reg,Pc}\rangle\hookrightarrow\langle\text{H,Cntr,St}',\text{Reg,Pc}'\rangle}$$

$$\frac{\begin{array}{l}\text{St (Cntr)} = \textbf{null}\\ \mathit{getStateOnExcRT}(<\text{H, Cntr, St, Reg, Pc}>,\ \texttt{NullPntrExc}, \texttt{m.excHndlS}) = S\end{array}}{\text{m}\vdash\text{arraylength:}\langle\text{H,Cntr,St,Reg,Pc}\rangle\hookrightarrow S}$$

Figure 2.4: Operational semantics for object manipulation

Fig. 2.4 and 2.5 give the semantics for instructions manipulating the program heap. Let us first focus on Fig. 2.4 which shows how object fields are accessed and modified. For instance, the instruction putfieldpops the top value contained on the stack top St (Cntr) and the reference value contained in St (Cntr - 1) are popped from the operand stack. If St (Cntr - 1) is not **null** 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.

The instruction for an access of the a value of a particular field for a particular object reference is the instruction getfield. The instruction pops the top stack element St (Cntr). If St (Cntr) is not **null** the value of the field `f` in the object referenced by the reference contained in St (Cntr), is fetched and pushed onto the operand stack St (Cntr). If St (Cntr) is **null** then a `NullPntrExc` is thrown, i.e. the stack counter is set to 0, a new object of type `NullPntrExc` is created in the memory heap store H and a reference to it is pushed onto the operand stack.

The instruction type_astore stores the value in St (Cntr) at index St (Cntr - 1) in the array St (Cntr - 2). 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. 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.

say what assignment compatible is

The instruction type_aload 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 *RefTypeCl* 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.

The instruction arraylength gets the length of an array. 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. Here we can see how the array length is modeled via the special object field arrLength.

Let us now look at Fig. 2.5 to see how object creation is modeled.

For example, a new class instance is created by the instruction new. A new fresh location `ref` is added in the memory heap H of type C, the stack counter Cntr is incremented. The reference `ref` is put on the stack top St (Cntr + 1). It deserves to note that although the semantics of the instance creation

described here is very close to the JVM semantics, we omit the so called VM errors, e.g. such the class from which an instance must be created is not found.

The instruction newarraycreates 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.

$$\frac{\begin{array}{l}(\mathrm{H}', \texttt{ref}) = \mathrm{newRef}(\mathrm{H}, C)\\ \mathrm{Cntr}' = \mathrm{Cntr} + 1\\ \mathrm{St}' = \mathrm{St}[\oplus \mathrm{Cntr} + 1 \rightarrow \texttt{ref}]\\ \mathrm{Pc}' = \mathrm{Pc} + 1\end{array}}{\mathrm{m} \vdash \text{new } C{:}\langle \mathrm{H},\mathrm{Cntr},\mathrm{St},\mathrm{Reg},\mathrm{Pc}\rangle \hookrightarrow \langle \mathrm{H}',\mathrm{Cntr}',\mathrm{St}',\mathrm{Reg},\mathrm{Pc}'\rangle}$$

$$\frac{\begin{array}{l}\mathrm{St}\ (\ \mathrm{Cntr}\ ) \geq 0\\ (\mathrm{H}', \texttt{ref}) = \mathrm{newArrRef}(\mathrm{H}, \mathit{type}, \mathrm{St}\ (\ \mathrm{Cntr}\ ))\\ \mathrm{Cntr}' = \mathrm{Cntr} + 1\\ \mathrm{St}' = \mathrm{St}[\oplus \mathrm{Cntr} + 1 \rightarrow \texttt{ref}]\\ \mathrm{Pc}' = \mathrm{Pc} + 1\end{array}}{\texttt{m}\vdash \text{newarray } \texttt{T}{:}\langle \mathrm{H},\mathrm{Cntr},\mathrm{St},\mathrm{Reg},\mathrm{Pc}\rangle \hookrightarrow \langle \mathrm{H}',\mathrm{Cntr}',\mathrm{St}',\mathrm{Reg},\mathrm{Pc}'\rangle}$$

$$\frac{\begin{array}{l}\mathrm{St}\ (\ \mathrm{Cntr}\ ) < 0\\ \mathit{getStateOnExcRT}(< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >,\ \texttt{NegArrSizeExc}, \texttt{m.excHndlS}) = S\end{array}}{\texttt{m}\vdash \text{newarray } \texttt{T}{:}\langle \mathrm{H},\mathrm{Cntr},\mathrm{St},\mathrm{Reg},\mathrm{Pc}\rangle \hookrightarrow S}$$

Figure 2.5: Operational semantics for object creation

Our language also supports instructions for checking if an object is of a given type. They are given in Fig. 2.6. For instance, the instruction instanceofchecks if the stack top element is of subtype C, then the 1 is pushed on the stack. If the object reference is **null**or not a subtype of C then 0 is pushed on the stack top. The checkcast instruction has a similar behavior, only that in case that the stack top element is not a subclass of C a `CastExc` is thrown.

The language presented here allows also to force exception throwing. This is done via the instruction athrow presented in Fig. 2.7. The stack top element must be a reference of an object of type `Throwable`. If the exception object St (Cntr) on the stack top is not **null** then there are two possible execution of the instruction. Either there is not an exception handler that protects this bytecode instruction from the exception type and the current method `m` terminates exceptionally by throwing the exceptio object St (Cntr) or there is a handler that protects this bytecode instruction from the exception thrown and the control is transfered to the instruction at index Pc^{eH} at which the exception handler starts. If the object on the stack top is **null**, a `NullPntrExc` is thrown and is handled as the function *getStateOnExcRT* prescribes.

Finally, in Fig. 2.8, we can see the semantics of method invokation. The first top $\mathit{meth}.\mathsf{nArgs}$ elements in the operand stack St are popped from the operand stack. If St (Cntr - $\mathit{meth}.\mathsf{nArgs}$) is not **null**, the invoked method is executed

$$\frac{\begin{array}{l}\mathsf{subtype}(\text{H.}\mathsf{TypeOf}\ (\text{St}\ (\ \text{Cntr}\ )), C)\\ \text{St}' = \text{St}[\oplus\text{Cntr} \rightarrow 1]\\ \text{Pc}' = \text{Pc} + 1\end{array}}{\text{instanceof}\ C{:}\langle\text{H,Cntr,St,Reg,Pc}\rangle \hookrightarrow \langle\text{H,Cntr,St}',\text{Reg,Pc}'\rangle}$$

$$\frac{\begin{array}{l}\neg(\mathsf{subtype}(\text{H.}\mathsf{TypeOf}\ (\text{St}\ (\ \text{Cntr}\ )), C)) \vee \text{St}\ (\ \text{Cntr}\ ) = \textbf{null}\\ \text{St}' = \text{St}[\oplus\text{Cntr} \rightarrow 0]\\ \text{Pc}' = \text{Pc} + 1\end{array}}{\texttt{m}\vdash\text{instanceof}\ \texttt{C}{:}\langle\text{H,Cntr,St,Reg,Pc}\rangle \hookrightarrow \langle\text{H,Cntr,St}',\text{Reg,Pc}'\rangle}$$

$$\frac{\begin{array}{l}\mathsf{subtype}(\text{H.}\mathsf{TypeOf}\ (\text{St}\ (\ \text{Cntr}\ )), C) \vee \text{St}\ (\ \text{Cntr}\ ) = \textbf{null}\\ \text{Pc}' = \text{Pc} + 1\end{array}}{\texttt{m}\vdash\text{checkcast}\ C{:}\langle\text{H,Cntr,St,Reg,Pc}\rangle \hookrightarrow \langle\text{H,Cntr,St,Reg,Pc}'\rangle}$$

$$\frac{\begin{array}{l}\neg(\mathsf{subtype}(\text{H.}\mathsf{TypeOf}\ (\text{St}\ (\ \text{Cntr}\ )), C))\\ \mathit{getStateOnExcRT}(<\text{H, Cntr, St, Reg, Pc}>, \texttt{CastExc}, \texttt{m.excHndlS}) = S\end{array}}{\texttt{m}\vdash\text{checkcast}\ C{:}\langle\text{H,Cntr,St,Reg,Pc}\rangle \hookrightarrow S}$$

Figure 2.6: OPERATIONAL SEMANTICS FOR TYPE CHECKING

$$\frac{\begin{array}{l}\text{St}\ (\ \text{Cntr}\ ) \neq \textbf{null}\\ \mathit{findExcHandler}(\text{H.}\mathsf{TypeOf}\ (\text{St}\ (\ \text{Cntr}\ )), \text{Pc}, \texttt{m.excHndlS}) = \bot\end{array}}{\texttt{m}\vdash\text{athrow:}\langle\text{H,Cntr,St,Reg,Pc}\rangle \hookrightarrow \langle\text{H,St}\ (\ \text{Cntr}\ )\rangle^{exc}}$$

$$\frac{\begin{array}{l}\text{St}\ (\ \text{Cntr}\ ) \neq \textbf{null}\\ \mathit{findExcHandler}(\text{H.}\mathsf{TypeOf}\ (\text{St}\ (\ \text{Cntr}\ )), \text{Pc}, \texttt{m.excHndlS}) = \text{Pc}^{eH}\\ \text{St}' = \text{St}[\oplus 0 \rightarrow \text{St}\ (\ \text{Cntr}\ )]\end{array}}{\texttt{m}\vdash\text{athrow:}\langle\text{H,Cntr,St,Reg,Pc}\rangle \hookrightarrow \langle\text{H,0,St}',\text{Reg,Pc}^{eH}\rangle}$$

$$\frac{\begin{array}{l}\text{St}\ (\ \text{Cntr}\ ) = \textbf{null}\\ \mathit{getStateOnExcRT}(<\text{H, Cntr, St, Reg, Pc}>, \texttt{NullPntrExc}, \texttt{m.excHndlS}) = S\end{array}}{\texttt{m}\vdash\text{athrow:}\langle\text{H,Cntr,St,Reg,Pc}\rangle \hookrightarrow S}$$

Figure 2.7: OPERATIONAL SEMANTICS FOR PROGRAMMATIC EXCEPTIONS

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.

$$\frac{\begin{array}{l}\text{St (Cntr} - meth.\mathsf{nArgs}\text{)} \neq \textbf{null} \\ meth :< \text{H}, 0, [\,], [\text{St (Cntr} - meth.\mathsf{nArgs}\text{)}, \ldots, \text{St (Cntr)}], 0 >\Rightarrow< \text{H}', \text{Res} >^{norm} \\ \text{Cntr}' = \text{Cntr} - \mathtt{m}.\mathsf{nArgs} + 1 \\ \text{St}' = \text{St}[\oplus\text{Cntr}' \rightarrow \text{Res}] \\ \text{Pc}' = \text{Pc} + 1\end{array}}{\mathtt{m}\vdash\text{invoke } meth{:}<\text{H,Cntr,St,Reg,Pc}>\hookrightarrow<\text{H}',\text{Cntr}',\text{St}',\text{Reg,Pc}'>}$$

$$\frac{\begin{array}{l}\text{St (Cntr} - meth.\mathsf{nArgs}\text{)} \neq \textbf{null} \\ meth :< \text{H}, 0, [\,], [\text{St (Cntr} - meth.\mathsf{nArgs}\text{)}, \ldots, \text{St (Cntr)}], 0 >\Rightarrow< \text{H}', \text{Exc} >^{exc} \\ findExcHandler(\text{H}'.\mathsf{TypeOf}\ (\text{Exc}), \text{Pc}, \mathtt{m}.\mathsf{excHndlS}) = \bot\end{array}}{\mathtt{m}\vdash\text{invoke } meth{:}<\text{H,Cntr,St,Reg,Pc}>\hookrightarrow<\text{H}',\text{Exc}>^{exc}}$$

$$\frac{\begin{array}{l}\text{St (Cntr} - meth.\mathsf{nArgs}\text{)} \neq \textbf{null} \\ meth :< \text{H}, 0, [\,], [\text{St (Cntr} - meth.\mathsf{nArgs}\text{)}, \ldots, \text{St (Cntr)}], 0 >\Rightarrow< \text{H}', \text{Exc} >^{exc} \\ findExcHandler(\text{H}'.\mathsf{TypeOf}\ (\text{Exc}), \text{Pc}, \mathtt{m}.\mathsf{excHndlS}) = \text{Pc}^{eH} \\ \text{St}' = \text{St}[\oplus 0 \rightarrow \text{Exc}]\end{array}}{\mathtt{m}\vdash\text{invoke } meth{:}<\text{H,Cntr,St,Reg,Pc}>\hookrightarrow<\text{H}',0,\text{St}',\text{Reg,Pc}^{eH}>}$$

$$\frac{\begin{array}{l}\text{St (Cntr} - meth.\mathsf{nArgs}\text{)} = \textbf{null} \\ getStateOnExcRT(< \text{H, Cntr, St, Reg, Pc} >,\ \mathtt{NullPntrExc}, \mathtt{m}.\mathsf{excHndlS}) = S\end{array}}{\mathtt{m}\vdash\text{invoke } meth{:}<\text{H,Cntr,St,Reg,Pc}>\hookrightarrow S}$$

Figure 2.8: OPERATIONAL SEMANTICS FOR PROGRAMMATIC EXCEPTIONS

2.9 Representing bytecode programs as control flow graphs

This section will introduce a formalization of an unstructured program in terms of a control flow graph. The notion of a loop in a bytecode program will be also defined. Note that in the following, the control flow graph corresponds to a method body.

Recall from Section 2.3 that every method $\mathtt{m}$ has an array of bytecode instructions $\mathtt{m}.\mathsf{body}$. The $k-th$ instruction in the bytecode array $\mathtt{m}.\mathsf{body}$ is denoted with $\mathtt{m}.\mathsf{body}[k]$. A method entry point instruction is an instruction at which an execution of a method starts. We assume that a method body has exactly one entry point and this is the first element in the method body $\mathtt{m}.\mathsf{body}[0]$.

The array of bytecode instructions of a method $\mathtt{m}$ determine the control flow graph $G(V, \rightarrow)$ of method $\mathtt{m}$ in which the vertices are the instructions of the method body.

In Fig. 2.9 we can see the execution relation between instructions. Note first that we rather use the infix notation $\mathtt{m}.\mathsf{body}[j] \rightarrow \mathtt{m}.\mathsf{body}[k]$ instead of $(\mathtt{m}.\mathsf{body}[j], \mathtt{m}.\mathsf{body}[k]) \in \rightarrow$ The definition says that there is an edge between two vertices $\mathtt{m}.\mathsf{body}[j]$ and $\mathtt{m}.\mathsf{body}[k]$ if they may execute immediately one after another. We say that $\mathtt{m}.\mathsf{body}[j]$ is a predecessor of $\mathtt{m}.\mathsf{body}[k]$ and that $\mathtt{m}.\mathsf{body}[k]$ is a successor of $\mathtt{m}.\mathsf{body}[j]$. The definition states the return instruction does not have successors. If $\mathtt{m}.\mathsf{body}[j]$ is the jump instruction if_cond k then its successors are the instruction at index k in the method body $\mathtt{m}.\mathsf{body}[k]$ and the

instruction $\mathtt{m.body}[j+1]$. From the definition, we also get that every instruction which potentially may throw an exception of type `Exc` has as successor the first instruction of the exception handler that may handle the exception type `Exc`. For instance, a successor of the instruction putfield is the exception handler entry point which can handle the `NullPntrExc` exception. The possible successors of the instruction athrow are the entry point of any exception handler in the method `m`. .

We assume that the control flow graph of every method 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 control flow graph and the practice shows that even hand written code is usually reducible. However, there exist algorithms to transform a non reducible control flow graph into a reducible one. For more information on program control flow graphs, the curious reader may refer to [6]. The next definition identifies backedges in the reducible control flow graph (intuitively, the edge that goes 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 2.9.1 (Backedge Definition) *Assume we have the method* `m` *with body* $\mathtt{m.body}$ *which determine the control flow graph* $G(V, \rightarrow)$ *with entry point* $\mathtt{m.body}[0]$. *In such a graph G, we say that loopEntry is a loop entry instruction and loopEnd is a loop end instruction of the same loop if the following conditions hold:*

- *for every execution path P from* $\mathtt{m.body}[0]$ *to loopEnd such that* $P = \mathtt{m.body}[0] \rightarrow^+$ *loopEnd there exists a subpath which is a prefix of P* $subP = \mathtt{m.body}[0] \rightarrow^*$ *loopEntry such that* $loopEnd \notin subP$
- *there is a path in which loopEntry is executed immediately after the execution of loopEnd (loopEnd* $\rightarrow$ *loopEntry)*

We denote the execution relation between loopEnd and loopEntry with $loopEnd \rightarrow^l loopEntry$ *and we say that* $\rightarrow^l$ *is a loop backedge.*

Note that in [6] reducibility is defined in terms of the dominator relation. Although not said explicitely, the first condition in the upper definition corresponds to the dominator relation[1].

We illustrate the above definition with the control flow graph of the example from Fig. 3.1.In the figure, we rather show the execution relation between basic blocks which is a standard notion denoting a sequence of instructions which execute sequentially and where only the last one may be a jump and the first may be a target of a jump. The black edges represent a sequential execution relation, while dashed edges represent loop backedge, i.e. the edge which stands for the execution relation between a final instruction (instruction at index `18`)

[1] we decided to not introduce the standard definitions as it has several technical details for the exposition of which we would need more space and which are of not particular interest for the current thesis

$$\frac{\mathtt{m.body}[j]=\text{if\_cond } k}{\mathtt{m.body}[j]\rightarrow\mathtt{m.body}[k]}$$

$$\frac{\mathtt{m.body}[j]=\text{goto } k}{\mathtt{m.body}[j]\rightarrow\mathtt{m.body}[k]}$$

$$\frac{\mathtt{m.body}[j]=\text{putfield}\wedge \mathit{findExcHandler}(\ \mathtt{NullPntrExc},j,\mathtt{m.excHndlS})=k}{\mathtt{m.body}[j]\rightarrow\mathtt{m.body}[k]}$$

$$\frac{\mathtt{m.body}[j]=\text{putfield}\wedge \mathit{findExcHandler}(\ \mathtt{NullPntrExc},j,\mathtt{m.excHndlS})=k}{\mathtt{m.body}[j]\rightarrow\mathtt{m.body}[k]}$$

$$\frac{\mathtt{m.body}[j]=\text{getfield}\wedge \mathit{findExcHandler}(\ \mathtt{NullPntrExc},j,\mathtt{m.excHndlS})=k}{\mathtt{m.body}[j]\rightarrow\mathtt{m.body}[k]}$$

$$\frac{\mathtt{m.body}[j]=\text{type\_astore}\wedge \mathit{findExcHandler}(\ \mathtt{NullPntrExc},j,\mathtt{m.excHndlS})=k}{\mathtt{m.body}[j]\rightarrow\mathtt{m.body}[k]}$$

$$\frac{\mathtt{m.body}[j]=\text{type\_astore}\wedge \mathit{findExcHandler}(\ \mathtt{ArrIndBndExc},j,\mathtt{m.excHndlS})=k}{\mathtt{m.body}[j]\rightarrow\mathtt{m.body}[k]}$$

$$\frac{\mathtt{m.body}[j]=\text{type\_aload}\wedge \mathit{findExcHandler}(\ \mathtt{NullPntrExc},j,\mathtt{m.excHndlS})=k}{\mathtt{m.body}[j]\rightarrow\mathtt{m.body}[k]}$$

$$\frac{\mathtt{m.body}[j]=\text{type\_aload}\wedge \mathit{findExcHandler}(\ \mathtt{ArrIndBndExc},j,\mathtt{m.excHndlS})=k}{\mathtt{m.body}[j]\rightarrow\mathtt{m.body}[k]}$$

$$\frac{\mathtt{m.body}[j]=\text{invoke n}\wedge \mathit{findExcHandler}(\ \mathtt{NullPntrExc},j,\mathtt{m.excHndlS})=k}{\mathtt{m.body}[j]\rightarrow\mathtt{m.body}[k]}$$

$$\frac{\mathtt{m.body}[j]=\text{invoke n}\wedge\ \begin{array}{l}\forall \mathtt{Exc},\exists s,\mathtt{n.exceptions}[s]=\mathtt{Exc}\wedge\\ \mathit{findExcHandler}(\mathtt{Exc},j,\mathtt{m.excHndlS})=k\end{array}}{\mathtt{m.body}[j]\rightarrow\mathtt{m.body}[k]}$$

$$\frac{\mathtt{m.body}[j]=\text{athrow}\wedge\forall \mathtt{Exc},\mathit{findExcHandler}(\mathtt{Exc},j,\mathtt{m.excHndlS})=k}{\mathtt{m.body}[j]\rightarrow\mathtt{m.body}[k]}$$

$$\frac{\mathtt{m.body}[j]\neq\text{goto m.body}[j]\neq\text{return } k=j+1}{\mathtt{m.body}[j]\rightarrow\mathtt{m.body}[k]}$$

Figure 2.9: EXECUTION RELATION BETWEEN BYTECODE INSTRUCTIONS IN A CONTROL FLOW GRAPH

in the bytecode cycle and the entry instruction of the cycle (instruction at index 19). Note that the "back" in "backedge" stands for that the control flow goes back to an instruction through which the execution path has already passed

which does not forcingly mean that the orientation of the edge will be in a backwards direction in the graphical representation of the graph.

che pas si c'est util mais Lilian a dit que ce n'est pas claire

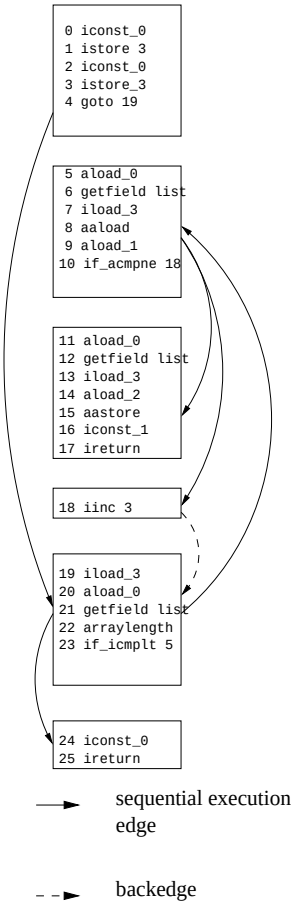

Figure 2.10: THE CONTROL FLOW GRAPH OF THE SOURCE PROGRAM FROM FIG.3.1

Chapter 3

Bytecode modeling language

This chapter presents the bytecode level specification language, called for short BML and a compiler from a subset of the high level Java specification language JML to BML which from now we shall call JML2BML. The chapter is organized as follows. In section 3.1, we give an overview of the main features of JML. A detailed overview of BML is given in section 3.3. As we stated before, we support also a compiler from the high level specification language JML into BML. The compilation process from JML to BML is discussed in section 3.5. The full specification of the new user defined Java attributes in which the JML specification is compiled is given in the appendix.

3.1 Overview of JML

JML [34] (short for Java Modeling Language) is a behavioral interface specification language tailored to Java applications which follows the design-by-contract approach (see [16]).

Over the last few years, JML has become the de facto specification language for Java source code programs. Several case studies have demonstrated that JML can be used to specify realistic industrial examples, and that the different tools allow to find errors in the implementations (see e.g. [17]). One of the reasons for its success is that JML uses a Java-like syntax. Other important factors for the success of JML are its expressiveness and flexibility.

JML is supported by several verification tools. Originally, it has been designed as a language of the runtime assertion checker [23] created by G.T. Leavens and . The JML runtime assertion checker compiles both the Java code and the JML specification into executable bytecode and thus, in this case, the verification consists in executing the resulting bytecode. Several static checkers based on formal logic exist which use JML as a specification language. Esc/java [46]

who are the others

whose first version used a subset of JML [1] is among the first tools supporting JML. Among the static checkers with JML are the Loop tool developed by the Formal group at the University of Nijmegen, the Jack tool developed at Gemplus, the Krakatoa tool created by the Coq group at Inria, France. The tool Daikon [28] tool uses a subset of JML for detecting loop invariants by run of programs. A detailed overview of the tools which support JML can be found in [19].

Specifications in JML are written using different predicates which are side-effect free Java expressions, extended with specification-specific keywords. 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 the Java syntax with few keywords and operators. For introducing method precondition and postcondition the keywords **requires** and **ensures** are used respectively, **modifies** keyword introduces the locations that can be modified by the method, **loop_invariant** stands for a loop invariant, the **loop_modifies** keyword gives the locations modified by a loop etc. The latter is not standard in JML and is an extension introduced in [20]. 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.

Fig. 3.1 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. Thus the method specification between lines 5 and 9 which exposes the method contract states the following. First the precondition (line 5) requires from any caller to assure that the instance variable `list` is not `null`. The frame condition (line 6) states that the method may only modify any of the elements in the instance field `list`. The method postcondition (lines 7—9) states the method will return `true` only if the replacement has been done. The method body contains a loop (lines 17—22) which is specified with a loop fame condition and a loop invariant (lines 13—16). The loop invariant (lines 14—16) says that all the elements of the list that are inspected up to now are different from the parameter object `obj1` as well as the local variable `i` is a valid index in the array `list`. The loop frame condition (line 13) states that only the local variable `i` and any element of the array field `list` may be modified in the loop.

JML also allows the declaration of special JML variables, that are used only for specification purposes. These variables are declared in comments with the **ghost** modificator and may be used only in specification clauses. Those variables can also be assigned. Ghost variables are usually used for expressing properties which can not be expressed with the program variables.

Fig. 3.2 is an example for how ghost variables are used. The example shows the class `Transaction` which manages transactions in the program. The class

[1] the current version of the tool esc/java 2 supports almost all JML constructs

```
1  public class ListArray {
2
3    private Object[] list;
4
5    //@ requires list != null;
6    //@ modifies list[*];
7    //@ ensures \result ==(\exists int i;
8    //@          0 <= i && i < list.length &&
9    //@          \old(list[i]) == obj1 && list[i] == obj2);
10   public boolean replace(Object obj1,Object obj2){
11     int i = 0;
12
13     //@ loop_modifies i, list[*];
14     //@ loop_invariant i <= list.length && i >=0
15     //@ &&(\forall int k;0 <= k && k < i ==>
16     //@     list[k] != obj1  && list[k] == \old(list[k]));
17     for (i = 0; i < list.length; i++){
18       if ( list[i] == obj1){
19         list[i] = obj2;
20         return true;
21       }
22     }
23     return false;
24   }
25 }
```

Figure 3.1: CLASS ListArray WITH JML ANNOTATIONS

is provided with a method for opening transactions `beginTransaction` and a method for closing transactions (`commitTransaction`). The specification declares a ghost variable `TRANS` (line 3) which keeps track if there is a running transaction or not, i.e. if the value of `TRANS` is 0 then there is no running transaction and if it has value 1 then there is a running transaction. The specification of the methods `beginTransaction` and `commitTransaction` models the property for no nested transactions. Thus, when the method `beginTransaction` is invoked the precondition (line 5) requires that there should be no running transaction and when the method is terminated the postcondition guarantees (line 6) that there is already a transaction running. We can also remark that the variable `TRANS` is set to its new value (line 8) in the body `beginTransaction`. Note that this high level property is difficult to express without the presence of the ghost variable `TRANS`.

A useful feature of JML is that it allows two kinds of method specification, a *light* and *heavy* weight specification. An example for a *light* specification is the annotation of method `replace` (lines 5—9) in Fig. 3.1. The specification

```
1  public class Transaction {
2
3    //@ ghost static private int TRANS = 0;
4
5    //@ requires TRANS == 0;
6    //@ ensures TRANS == 1;
7    public void  beginTransaction() {
8      //@ set TRANS = 1;
9        ...
10   }
11
12   //@ requires TRANS == 1;
13   //@ ensures TRANS == 0;
14   public void  commitTransaction() {
15     //@ set TRANS = 0;
16       ...
17   }
18
19 }
```

Figure 3.2: specifying No Nested Transaction property with ghost variable

in the example states what is the expected behavior of the method and under what conditions it might be called. The user, however in JML, has also the possibility to write very detailed method specifications. This style of specification is called a *heavy* weight specification. It is introduced by the JML keywords **normal_behavior** and **exceptional_behavior**. As the keywords suggest every of them specifies a specific normal or exceptional behavior of a method. (see [44]).

The keyword **normal_behavior** introduces a precondition, frame condition and postcondition such that if the precondition holds in the prestate of the method then the method will terminate normally and the postcondition will hold in the poststate. Note that this clause guarantees that the method will not terminate on an exception and thus the exceptional postcondition for any kind of exception (i.e. for the exception class `Exception`) is **false**. An example for a *heavy* weight specification is given in Fig. 3.3. In the figure, method `divide` has two behaviors, one in case the method terminates normally (lines 11—14) and the other (lines 17—20) in case the method terminates by throwing an object reference of `ArithmeticException`. In the normal behavior case, the exceptional postcondition is ommitted specification as by default if the precondition (line 12) holds this assures that no exceptional termination is possible. Another observation over the example is that the exceptional behavior is introduced with the JML keyword **also**. The keyword **also** serves for introducing every new be-

```
1  public class C {
2      int a;
3
4      //@ public instance invariant a > 0 ;
5
6      //@ requires val > 0 ;
7      public C(int val){
8          a = val ;
9      }
10
11     //@ public normal_behavior
12     //@ requires b > 0;
13     //@ modifies a;
14     //@ ensures  a == \old(a) / b;
15     //@
16     //@ also
17     //@ public exceptional_behavior
18     //@ requires b == 0;
19     //@ modifies \nothing;
20     //@ exsures (ArithmeticException) a == \old(a);
21     public void divide(int b) {
22         a = a / b;
23     }
24 }
```

Figure 3.3: An example for a method with a heavy weight specification in JML

havior of a method except the first one. Note that the keyword **also** is used in case a method overrides a method from the super class. In this case, the method specification (*heavy* or *light* weight) is preceded by the keyword **also** to indicate that the method should respect also the specification of the super method.

JML can be used to specify not only methods but also properties of a class or interfaces. A Java class may be specified with an invariant or history constraints. An invariant of a class is a predicate which holds at all visible states of every object of this class (see for the definition of visible state in the JML reference manual [26]). An invariant may be either static (i.e. talks only about static fields) or instance (talks about instance fields). A Class history constraints is a property which relates the initial and terminal state of every method in the corresponding class. The class `C` in Fig.3.3 has also an instance invariant which states that the instance variable `a` is always greater than `0`.

3.2 Design features of BML

Before proceeding with the syntax and semantics of BML, we would like to discuss the design choices made in the language. Particularly, we will see what are the benefits of our approach as well as the restrictions that we have to adopt. Now, we focus on the desired features of BML, how they compare to JML and what are the motivations that led us to these decisions:

Java compiler independance Class files containing BML specification must not depend on any non optimizing compiler.

To do this, the process of the Java source compilation is separate from the JML compilation. More particularly, the JML2BML(short for the compiler from JML to BML) compiler takes as input a Java source file annotated with JML specification and its Java class produced by a non optimizing compiler containing a debug information.

JVM compatibility The class files augmented with the BML specification must be executable by any implementation of the JVM specification. Because the JVM specification does not allow inlining of any user specific data in the bytecode instructions BML annotations must be stored separately from the method body (the list of bytecode instructions which represents its body).

In particular, the BML specification is written in the so called user defined attributes in the class file. The JVM specification defines the format of those attributes and mandates that any user specific information should be stored in such attributes. Note, that attribute which encodes the specification referring to a particular bytecode instruction contains information about the index of this instruction. For instance, BML loop invariants are stored in a user defined attribute in the class file format which contains the invariant as well as the index of the entry point instruction of the loop.

Thus, BML encoding is different from the encoding of JML specification where annotations are written directly in the source text as comments at a particular point in the program text or accompany a particular program structure. For instance, in Fig. 3.1 the reader may notice that the loop specification refers to the control structure which follows after the specification and which corresponds to the loop. This is possible first because the Java source language is structured, and second because writing comments in the source text does not violate the Java or the JVM specifications.

BML corresponds to a partially desugared version of JML BML is designed to correspond to a subset of the desugared version of JML. We consider that such encoding makes the verification procedure more efficient. Let us see why. Because BML corresponds to a desugared version of JML, this means that on verification time the BML specification does not need much processing and thus, it can be easily translated to the data

structures used in the verification scheme. This makes BML suitable for verification on devices with limitted resources.

We impose also few restrictions on the structure of the class file:

Line_Number_Table and Local_Variable_Table A requirement to the class file format is that it must contain the **Line_Number_Table** and **Local_Variable_Table** attributes. The presence in the Java class file format of these attribute is optional [49], yet almost all standard non optimizing compilers can generate these data. The **Line_Number_Table** is part of the compilation of a method and describes the link between the Java source lines and the Java bytecode. The **Local_Variable_Table** describes the local variables that appear in a method. These attributes are usually used by debuggers as they describe the relation between source and bytecode. It is also necessary for the compiler from JML to BML, as we shall see later in Section 3.5.

Non-optimizing compilation The compilation process from JML to BML relies on finding the relation between source and its compilation into bytecode. As code optimization can make this relation to disappear, we require that the bytecode be produced by a non-optimizing compiler.

3.3 The subset of JML supported in BML

BML corresponds to a representative subset of JML and is expressive enough for most purposes including the description of non trivial functional and security properties. The following Section 3.3.1 gives the notation conventions adopted here and Section 3.3.2 gives the formal grammar of BML as well as an informal description of its semantics.

3.3.1 Notation convention

- Nonterminals are written with a *italics* font
- Terminals are written with a **boldface** font
- brackets [ ] surround optional text.

3.3.2 BML Grammar

$constants_{bml}$ ::= *intLiteral* | *signedIntLiteral* | **null** | *ident*

signedIntLiteral ::= $+$*nonZerodigit*[*digits*] | $-$*nonZerodigit*[*digits*]

intLiteral ::= *digit* | *nonZerodigit*[*digits*]

digits ::= *digit*[*digits*]

digit ::= **0** | *nonZerodigit*

nonZerodigit ::= **1** | ... | **9**

ident ::= # *intLiteral*

boundVar ::= **bv_***intLiteral*

E ::= $constants_{bml}$
| **reg**(*digits*)
| *E*.*ident*
| *ident*
| **arrayAccess**(*E*, *E*)
| *E op E*
| **cntr**
| **st**(*E*)
| **old**(*E*)
| **EXC**
| **result**
| *boundVar*
| **typeof**(*E*)
| **type**(*ident*)
| **elemtype**(*E*)
| **TYPE**

op ::= **+** | **-** | **mult** | **div** | **rem**

$\mathcal{R}$ ::= $=|\neq|\leq|\leq|\geq|>|<$:

P ::= *E* $\mathcal{R}$ *E*
| **true**
| **false**
| *not P*
| $P \wedge P$
| $P \vee P$
| $P \Rightarrow P$
| $P \iff P$
| $\forall$ *boundVar*, *P*
| $\exists$ *boundVar*, *P*

classSpec ::= **invariant** *modifier P*
| **classConstraint** *P*
| **declare ghost** *ident ident*

modifier ::= **instance** | **static**

intraMethodSpec ::= **atIndex** *nat*;
assertion;

assertion ::= *loopSpec*

$$
\begin{array}{lll}
\textit{methodSpec} & ::= & \textit{specCase} \\
 & | & \textit{specCase}\ \textbf{also}\ \textit{methodSpec} \\
 & & \\
\textit{specCase} & ::= & \{| \\
 & & \quad \textbf{requires}\ P; \\
 & & \quad \textbf{modifies}\ \textit{list locations}; \\
 & & \quad \textbf{ensures}\ P; \\
 & & \quad \textit{exsuresList} \\
 & & |\} \\
 & & \\
\textit{exsuresList} & ::= & [] \mid \textbf{exsures}\ (\textit{ident})\ P; \textit{exsuresList} \\
 & & \\
\textit{locations} & ::= & E.\textit{ident} \\
 & | & \textbf{reg}(i) \\
 & | & \textit{arrayModAt}(E, \textit{specIndex}) \\
 & | & \textbf{everything} \\
 & | & \textbf{nothing} \\
 & & \\
\textit{specIndex} & ::= & \text{all} \mid i_1..i_2 \mid i
\end{array}
$$

3.3.3 Syntax and semantics of BML

In the following, we will discuss informally the semantics of the syntax structures of BML. Note that most of them have an identical counterpart in JML and their semantics in both languages is the same. In the following, we will concentrate more on the specific syntactic features of BML and will just briefly comment the BML features which it inherits from JML like for instance, preconditions and which we have mentioned already[2].

BML expressions

Among the common features of BML and JML are the following expressions: field access expressions $E.\textit{ident}$, array access ($\textbf{arrayAccess}(E^1, E^2)$), arithmetic expressions ($E\ op\ E$). Like JML, BML may talk about expression types. As the BML grammar shows, $\backslash\textbf{typeof}(E)$ denotes the dynamic type of the expression E, $\backslash\textbf{type}(\textit{ident})$ is the class described at index *ident* in the constant pool of the corresponding class file. The construction $\backslash\textbf{elemtype}(E)$ denotes the type of the elements of the array E, and $\backslash\textbf{TYPE}$, like in JML, stands for the Java type `java.lang.Class`.

However, expressions in JML and BML differ in the syntax more particularly this is true for identifiers of local variables, method parameters, field and class

[2]because we have already discussed in Section 3.1 the JML constructs for pre and post-conditions, loop invariants, operators like **old**, **result**, etc. we would not return to them anymore as their semantics is exactly the same as the one of JML

identifiers. In JML, all these constructs are represented syntactically by their names in the Java source file. This is not the case in BML.

We first look at the syntax of method local variables and parameters. The class file format stores information for them in the array of local variables. That is why, both method parameters and local variables are represented in BML with the construct **reg**(i) which refers to the element at index i in the array of local variables of a method. Note that the `this` expression in BML is encoded as **reg**(0). This is because the reference to the current object is stored at index 0 in the array of local variables.

Field and class identifiers in BML are encoded by the respective number in the constant pool table of the class file. For instance, the syntax of field access expressions in BML is $E.ident$ which stands for the value in the field at index $ident$ in the class constant pool for the reference denoted by the expression E.

The BML grammar defines the syntax of identifiers differently from their usual syntax. Particularly, in BML those are positive numbers preceded by the symbol # while usually the syntax of identifiers is a chain of characters which always starts with a letter. The reason for this choice in BML is that identifiers in BML are indexes in the constant pool table of the corresponding class.

Fig.3.4 gives the bytecode as well as the BML specification of the code presented in Fig.3.3. As we can see, the names of the local variables, field and class names are compiled as described above. For instance, at line 3 in the specification we can see the precondition of the first specification case. It talks about **reg**(1) which is the element in the array of local variables of the method and which is the compilation of the method parameter `b` (see Fig. 3.3).

About the syntax of class names, after the **exsures** clause at line 5 follows a BML identifier (#25) enclosed in parenthesis. This is the constant pool index at which the Java exception type `java.lang.Exception` is declared.

A particular feature of BML is that it supports stack expressions which do not have a counterpart in JML. These expressions are related to the way in which the virtual machine works, i.e. we refer to the stack and the stack counter. Because intermediate calculations are done by using the stack, often we will need stack expressions in order to characterise the states before and after an instruction execution. Stack expressions are represented in BML as follows:

- **cntr** represents the stack counter.
- **st**(E) stands for the element in the operand stack at position E. For instance, the element below the stack top is represented with **st**(**cntr** − 1) Note that those expressions may appear in predicates that refer to intermediate instructions in the bytecode.

BML predicates

The properties that our bytecode language can express are from first order predicate logic. The formal grammar of the predicates is given by the nonterminal P. From the formal syntax, we can notice that BML supports the standard

```
1
2  Class instance invariant:
3     lv(0).#19 > 0;
4
5
6  Method specification:
7     {|
8      requires lv(1) > 0;
9      modifies lv(0).#19;
10     ensures  lv(0).#19 == \old( lv(0).#19 ) / lv(1);
11     exsures  ( #25 ) false;
12    |}
13    also
14    {|
15     requires lv(1) == 0;
16     modifies \nothing;
17     ensures false;
18     exsures ( #26 ) lv(0).#19  == \old(lv(0).#19);
19    |}
20
21  public void divide(int lv(1))
22        0 aload 0
23        1 dup
24        2 getfield #19 // instance field a
25        3 iload 1
26        4 idiv
27        5 putfield #19 // instance field a
28        6 return
```

Figure 3.4: An example for a heavy weight specification in BML

logical connectors $\wedge, \vee, \Rightarrow$, existential $\exists$ and universal quantification $\forall$ as well as standard relation between the expressions of our language like $\neq, =, \leq, \leq \ldots$

Class Specification

The nonterminal *classSpec* in the BML grammar defines syntax constructs for the support of class specification. Note that these specification features exist in JML and have exactly the same semantics. However, we give a brief description of the syntax. Class invariants are introduced by the terminal **invariant**, history constraints are introduced by the terminal **classConstraint**. For instance, in Fig. 3.4 we can see the BML invariant resulting from the compilation of the JML specification in Fig. 3.3.

Like JML, BML supports ghost variables. As we can notice in the BML

grammar, their syntax in the grammar is **declare ghost** *ident ident*. The first *ident* is the index in the constant pool which contains a description of the type of the ghost field. The second *ident* is the index in the constant pool which corresponds to the name of the ghost field.

Frame conditions

BML supports frame conditions for methods and loops. These have exactly the same semantics as in JML. The nonterminal that defines the syntax for frameconditions is *locations*. We look now what are the syntax constructs that may appear in the frame condition:

- $E.ident$ states that the method or loop modifies the value of the field at index *ident* in the constant pool for the reference denoted by E
- **reg**(i) states that the local variable may modified by a loop. Note that this kind of modified expression makes sense only for expressions modified in a loop. Indeed, a modification of a local variable does not make sense for a method frame condition, as methods in Java are called by value, and thus, a method can not cause a modification of a local variable that is observable from the outside of the method.
- $arrayModAt(E, specIndex)$ states that the components at the indexes specified by *specIndex* in the array denoted by E may be modified. The indexes of the array components that may be modified *specIndex* have the following syntax:
  - i is the index of the component at index i. For instance, $arrayModAt(E, i)$ means that the array component at index i might be modified.
  - all specifies that all the components of the array may be modified, i.e. the expression $arrayModAt(E, \text{all})$ means that any element in the array may potentially be modified.
  - $i_1..i_2$ specifies the interval of array components between the index i_1 and i_2.
- **everything** states that every location might be modified by the method or loop
- **nothing** states that no location might be modified by a method or loop

Inter — method specification

In this subsection, we will focus on the method specification which is visible by the other methods in the program or in other words the method pre, post and frame conditions. The syntax of those constructs is given by the nonterminal *methodSpec*. As their meaning is exactly the same as in JML and we have

already discussed the latter in Section 3.1, we shall not spend more lines here on those.

The part of the method specification which deserves more attention is the syntax of heavy weight method specification in BML. In Section 3.1, we saw that JML supports syntactic sugar for the definition of the normal and exceptional behavior of a method. The syntax BML does not support these syntactic constructs but rather supports their desugared version (see [62] for a detailed specification of the JML desugaring process). A specification in BML may declare several method specification cases like in JML. The syntactic structure of a specification case is defined by the nonterminal *specCase*.

We illustrate this with an example in Fig. 3.4. In the figure, we remark that BML does not have the syntactic sugar for normal and exceptional behavior. On the contrary, the specification cases now explicitely declare their behavior. The first specification case (the first bunch of specification enclosed in {| |}) corresponds to the **normal_behavior** specification case in the code from Fig. 3.3. Note that it does not have an analog for the JML keyword **normal_behavior** and that it declares explicitely what is the behavior of the method in this case, i.e. the exceptional postcondition is declared **false** for any exceptional termination.

The second specification case in Fig.3.4 corresponds to the **exceptional_behavior** case of the source code specification in Fig.3.3. It also specifies explicitely all details of the expected behavior of the method, i.e. the method postcondition is declared to be **false**.

Intra — method specification

As we can see from the formal grammar in subsection 3.3.2, BML allows to specify a property that must hold at particular program point inside a method body. The nonterminal which describes the grammar of assertions is *intraMethodSpec*. Note that a particularity of BML specification, i.e. loop specifications or assertion at particular program point contains information about the point in the method body at which it refers. For instance, the loop specification in BML given by the nonterminal *loopSpec* may contain apart from the loop invariant predicate (**loop_invariant**), the list of modified variables (**loop_modifies**) and the decreasing expression (**loop_decreases**) but also the index of the loop entry point instruction (**atIndex**).

We illustrate this with the example in Fig. 3.5 which represents the bytecode and the BML specification from the example in Fig. 3.1. The first line of the BML specification specifies that the loop entry is the instruction at index 19 in the array of bytecode instructions. The predicate representing the loop invariant introduced by the keword **loop_invariant** respects the syntax for BML expressions and predicates that we described above.

```
1
2
3  Loop specification :
4
5      atIndex 19;
6      loop_modifies lv(0).#19[*], lv(3);
7      loop_invariant
8        lv(3) >= 0 &&
9        lv(3) < lv(0).#19.arrLength &&
10       \forall  bv_1 ;
11                ( bv_1 >= 0 &&
12                bv_1 < lv(0).#19.arrLength ==>
13                     lv(0).#19[bv_1] != lv(1) )
14
15  public int replace(Object  lv(1),Object lv(2) )
16   0 const 0
17   1 store 3
18   2 const 0
19   3 store 3
20   4 goto 19
21   5 load 0
22   6 getfield #19 // instance field list
23   7 load 3
24   8 aaload
25   9 load 1
26  10 if_acmpne 18
27  11 load 0
28  12 getfield #19 // instance field list
29  13 load 3
30  14 load 2
31  15 aastore
32  16 const 1
33  17 return
34  18 iinc 3
35  19 load 3    // loop entry
36  20 load 0
37  21 getfield #19 // instance field list
38  22 arraylength
39  23 if_icmplt 5
40  24 const 0
41  25 return
```

Figure 3.5: An example for a loop specification in BML

3.4 Well formed BML specification

In the previous Section 3.3, we gave the formal grammar of BML. However, we are interested in a strict subset of the specifications that can be generated from this grammar. In particular, we want that a BML specification is well typed and respects structural constraints. The constraints that we impose here are similar to the type and structural constraints that the bytecode verifier imposes over the class file format.

Examples for type constraints that a valid BML specification must respect :

- the array expression **arrayAccess**(E^1, E^2) must be such that E^1 is of array type and E^2 is of integer type.
- the field access expression $E.ident$ is such that E is of subtype of the class where the field described by the constant pool element at index *ident* is declared
- For any expression $E^1 op E^2$, E^1 and E^2 must be of a numeric type.
- in the predicate $E^1 r E^2$ where $r = \leq, <, \geq, >$ the expressions E^1 and E^2 must be of numerical type.
- in the predicate $E^1 <: E^2$, the expressions E^1 and E^2 must be of type **\TYPE** (which is the same as `java.lang.Class`).
- the expression **\elemtype(**E**)** must be such that E has an array type.
- etc.

Example for structural constraint are :

- All references to the constant pool must be to an entry of the appropriate type. For example: the field access expression $E.ident$ is such that the *ident* must reference a field in the constant pool; or for the expression **\type(***ident***)**, *ident*must be a reference to a constant class in the constant pool
- every *ident* in a BML specification must be a correct index in the constant pool table.
- if the expression **reg**(i) appears in a method BML specification, then i must be a valid index in the array of local variables of the method

An extension of the bytecode verifier may perform the checks if a BML specification respects this kind of structural and type constraints. However, we have not worked on this problem and is a good candidate for a future work. For the curious reader, it will be certainly of interest to turn to the Java Virtual Machine specification [49] which contains the official specification of the Java bytecode verifier or to the existing literature on bytecode verification (see the overview article [47])

3.5 Compiling JML into BML

In this section, we turn to the JML2BML compiler. As we shall see, the compilation consists of several phases, namely compiling the Java source file, preprocessing of the JML specification, resolution and linking of names, locating the position of intra — method specification, processing of boolean expressions and finally encoding the BML specification in user defined class file attributes. (their structure is predefined by JVMS). In the following, we look in details at the phases of the compilation process:

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. Those attributes are important for the next phases of the JML compilation.

2. Compilation of Ghost field declarations
   JML specification is invisible by the Java compilers. Thus Java compilers omit the compilation of ghost variables declaration. That is why it is the responsibility of the JML2BML compiler to do this work. For instance, the compilation of the declaration of the ghost variable from Fig. 3.2 is given in Fig.3.6 which shows the data structure **Ghost_field_Attribute** in which the information about the field `TRANS` is encoded in the class file format. Note that, the constant pool indexes **#18** and **#19** which contain its description were not in the constant pool table of the class file `Transaction.class` before running the JML2BML compiler on it.

```
Ghost_field_Attribute {
    ...
    {  access_flag 10;
       name_ index = #18;
       descriptor_index = #19
    } ghost[1];
}
```

- **access_flag**: The kind of access that is allowed to the field
- **name_index**: The index in the constant pool which contains information about the source name of the field
- **descriptor_index**: The index in the constant pool which contains information about the name of the field type

Figure 3.6: Compilation of ghost variable declaration

3. Desugaring of the JML specification
   The phase consists in converting the JML method heavy-weight behaviours and the light - weight non complete specification into BML specification cases. It corresponds to part of the standard JML desugaring as described in [62]. For instance, the BML compiler will produce from the specification in Fig.3.3 the BML specification given in Fig.3.4

4. Linking with source data structures
   When the JML specification is desugared, we are ready for the linking and resolving phases. In this stage, the JML specification gets into an intermediate format in which the identifiers are resolved to their corresponding data structures in the class file. The 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.

   For instance, consider once again the example in Fig. 3.3 and more particularly the first specification case of method `divide` whose precondition `b > 0` contains the method parameter identifier `b`. In the linking phase, the identifier `b` is resolved to the local variable **reg**(1) in the array of local variables for the method `divide`. We have a similar situation with the postcondition `a == \old(a) / b` which mentions also the field `a` of the current object. The field name `a` is compiled to the index in the class constant pool which describes the constant field reference. The result of the linking process is in Fig.3.4.

   If, in the JML specification a field identifier appears for which no constant pool index exists, it is added in the constant pool and the identifier in question is compiled to the new constant pool index. This happens when declarations of JML ghost fields are compiled.

5. Locating the points for the intra —method specification

   In this phase 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 in the bytecode. For this, the **Line_Number_Table** attribute is used. The **Line_Number_Table** attribute describes the correspondence between the Java source line and the instructions of its respective bytecode. In particular, for every line in the Java source code the **Line_Number_Table** specifies the index of the beginning of the basic block[3] in the bytecode which corresponds to the source line. Note however, that a source line may correspond to more than one instruction in the **Line_Number_Table**.

[3] a basic block is a sequence of instructions which does not contain jumps except may be for the last instruction and neither contains target of jumps except for the first instruction. This notion comes from the compiler community and more information on this one can find at [6]

Line_Number_Table

start_pc	**line**
...	
2	**17**
18	**17**

Figure 3.7: **Line_Number_Table** FOR THE METHOD `replace` IN FIG. 3.1

This poses problems for identifying loop entry instruction of a loop in the bytecode which corresponds to a particular loop in the source code. For instance, for method `replace` in the Java source example in Fig. 3.1 the java compiler will produce two lines in the **Line_Number_Table** which correspond to the source line **17** as shown in Fig. 3.7. The problem is that none of the basic bloks determined by instructions **2** and **18** contain the loop entry instruction of the compilation of the loop at line **17** in Fig. 3.1. Actually, the loop entry instruction in the bytecode in Fig. 3.5 (remember that this is the compilation in bytecode of the Java source in Fig. 3.1) which corresponds to the in the bytecode is at index **19**.

Thus for identifying loop entry instruction corresponding to a particular loop in the source code, we use an heuristics. It consists in looking for the first bytecode loop entry instruction starting from one of the **start_pc** indexes (if there is more than one) corresponding to the start line of the source loop in the **Line_Number_Table**. The algorithm works under the assumption that the control flow graph of the method bytecode is reducible. This assumption guarantees that the first loop entry instruction found starting the search from an index in the **Line_Number_Table** corresponding to the first line of a source loop will be the loop entry corresponding to this source loop. However, we do not have a formal argumentation for this algorithm because it depends on the particular implementation of the compiler. From our experiments, the heuristic works successfully for the Java Sun non optimizing compiler.

une presentation tres laide

6. Compilation of the JML boolean expressions into BML

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 integral 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 `replace` and its specification

$$\backslash\mathbf{result} = 1$$

$$\Longleftrightarrow$$

$$\exists \mathbf{bv\_0}, \left(\begin{array}{l} 0 \leq \mathbf{bv\_0} \wedge \\ \mathbf{bv\_0} < len(\#19(\mathbf{reg}(0))) \wedge \\ \mathbf{arrayAccess}(\#19(\mathbf{reg}(0)), \mathbf{bv\_0}) = \mathbf{reg}(1) \end{array} \right)$$

Figure 3.8: The compilation of the postcondition in Fig. 3.1

in Fig.3.1 the postcondition states the equality between the JML expression **result** and a predicate. This is correct as the method `replace` 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 `replace` 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[4].

Finally, the compilation of the postcondition of method `replace` is given in Fig. 3.8. 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.3.1 is compiled into logical equivalence.

7. Encoding BML specification into user defined class attributes
   The specification expression and predicates are 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.

   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. 3.9. This attribute is an array of data structures each describing a single loop from the method source code. From the figure, we notice that every element describing the specification for a particular loop contains the index of the corresponding loop entry instruction **index**, the loop modifies clause (**modifies**), the loop invariant (**invariant**), an expression which guarantees termination (**decreases**).

[4] when generating proof obligations we add for every source boolean expression an assumption that it must be equal to 0 or 1. A reasonable compiler would encode boolean values in a similar 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 3.9: Structure of the Loop Attribute

Chapter 4

Assertion language for the verification condition generator

In this chapter we shall focus on a particular fragment of BML which will be extended with few new constructs. The part of BML in question is the assertion language that our verification condition generator manipulates as we shall see in the next Chapter 5.

The assertion language presented here will abstract from most of the BML specification clauses described in Section 3.3. Our interest will be focused only on method and loop specification. Some parts of BML will be completely ignored either for keeping things simple or because those parts are desugared and result into the BML fragment of interest. For instance, we do not consider here multiple method specification cases, neither assertions in particular program point for the first reason. The assertion language presented here discards also class invariants and history constraints because they boil down to method pre and postconditions.

The rest of this chapter is organized as follows. Section 4.1 presents what is exactly the BML fragment of interest and its extensions. Section 4.4 shows how we encode method and loop specification as well as presents a discussion how some of the ignored BML specification constructs are transformed into method pre and postconditions. The last two sections are concentrated on the formal meaning of the assertion language, i.e. Section 4.2 defines the substitution for the assertion language and Section 4.3 gives formal semantics of the assertion language.

4.1 The assertion language

The assertion language in which we are interested corresponds to the BML expressions (nonterminal E) and predicates (nonterminal P) extended with several new constructs. The extensions that we add are the following:

- Extensions to expressions. The assertion language that we present here must be suitable for the verification condition calculus. Because the verification calculus talks about updated field and array access we should be able to express them in the assertion language. Thus we extend the grammar of BML expression with the following constructs concerning update of fields and arrays :
  - update field access expression $f[\oplus E \rightarrow E](E)$.
  - update array access expression arrAccess$[\oplus(E, E) \rightarrow E](E, E)$

  The verification calculus will need to talk about reference values. Thus we extend the BML expression grammar to support reference values *RefVal*. Note that in the following integers **int** and *RefVal* will be referred to with *Values*.

- Extensions to predicates. Our bytecode language is object oriented and thus supports new object creation. Thus we will need a means for expressing that a new object has been created during the method execution.

  We extend the language of BML formulas with a new user defined predicate **instances**(*RefVal*). Informally, the semantics of the predicate **instances**(`ref`) where `ref` $\in$ *RefVal* means that the reference `ref` has been allocated when the current method started execution.

The assertion language will use the names of fields and classes for the sake of readability instead of their corresponding indexes in the constant pool as is in BML.

We would like to discuss in the following how and why BML constructs like class invariants and history constraints can be expressed as method pre and postconditions:

- Class invariants. A class invariant (**invariant**) is a property that must hold at every visible state of the class. This means that a class invariant must hold when a method is called and also must be established at the end of a method execution. A class invariant must be established in the poststate of the constructor of this class. Thus the semantics of a class invariant is part of the pre and postcondition of every method and is a part of the postcondition of the constructor of the class.

- History constraints. A class history constraint (**classConstraint**) gives a relation between the pre and poststate of every method in the class. A class history constraint thus can be expressed as a postcondition of every method in the class.

4.2 Substitution

Substitution is defined inductively in a standard way over the expression and formula structure. Still, we extend substitution to deal with field and array updates as follows:

$$E[f \backslash f[\oplus E \rightarrow E]]$$

This substitution does not affect any of the ground expressions,, i.e. it does not affect local variables ($\mathbf{reg}(i)$), the constants of our language (*constants*), the stack counter (**cntr**), the result expression (**result**), the thrown exception instance variable (**EXC**). For instance, the following substitution does not change $\mathbf{reg}(1)$:

$$\mathbf{reg}(1)[f \backslash f[\oplus E \rightarrow E]] = \mathbf{reg}(1)$$

Field substitution affects only field objects as we see in the following:

$$E.f_1[f_2 \backslash f_2[\oplus E_1 \longrightarrow E_2]] =$$

$$\begin{cases} E.f_1 & if\ f_1 \neq f_2 \\ E.f_2[\oplus E_1 \longrightarrow E_2] & else \end{cases}$$

$$E.f_1[\oplus E_1 \rightarrow E_2][f_2 \backslash f_2[\oplus E_3 \rightarrow E_4]] =$$

$$\begin{cases} f_1[\oplus E_1[f_2 \backslash f_2[\oplus E_3 \rightarrow E_4]] \rightarrow E_2[f_2 \backslash f_2[\oplus E_3 \rightarrow E_4]]] & if\ f_1 \neq f_2 \\ f_1\ \begin{array}{l}[\oplus E_1[f_2 \backslash f_2[\oplus E_3 \rightarrow E_4]] \longrightarrow E_2[f_2 \backslash f_2[\oplus E_3 \rightarrow E_4]]] \\ [\oplus E_3 \longrightarrow E_4]\end{array} & else \end{cases}$$

For example, consider the following substitution expression:

$$\mathbf{reg}(1).f[f \backslash f[\oplus \mathbf{reg}(2) \rightarrow 3]]$$

This results in the new expression :

$$\mathbf{reg}(1).f[\oplus \mathbf{reg}(2) \rightarrow 3]$$

The same kind of substitution is allowed for array access expressions, where the array object arrAccess can be updated.

4.3 Interpretation

In this section, we shall focus on the semantics of formulas and expressions w.r.t. a state configuration. Note that we shall give a semantics which ignores partiality in the language. Of course, this is an important theoretical question

which by itself have given rise to a lot of research and has received different answers. What we mean by partiality is the existence of functions like division or dereferencing of a field, array indexing etc. To get a precise idea of the problem we can consider the following logical statements:

$$\textit{(1)}\ f(E) == 3$$
$$\textit{(2)}\ \mathbf{arrayAccess}(E_1, E_2) == 5$$

Under certain conditions, these formulas may not have a meaning. In case *(1)* the statement does not make sense if E is **null**. In case *(2)* the statement does not make sense if either E_1 is **null** or E_2 is not in the bounds of E_1.

Building a logic for partiality is not trivial. Different solutions exist. Gries and Schnieder [32] give a solution to the problem which consists in function underspecification and thus avoid the problem of undefideness. More particularly, their approach considers all functions as total but for some value of the argument the function is not specified. B. Schieder and M. Broy [65] give an extension of the calculus of boolean structures proposed by Dijkstra and Scholten ([27]) with a third boolean value undefined where the logical connectives are extended in a nonmonotonic way preserves the properties of the classical logic on the price of having two sorts of equality. A naive three valued logic, where expressions may be evaluated to a value or undefined in a state and formulas might be false, true or undefined in a state appear to lose certain nice properties which standard logic with equality have as for instance associativity of equality, the excluded middle [32] etc. In [18], L. Burdy shows a three valued logic in which the above features of classical logic are preserved. a well-definedness operator $\delta() : E \cup P \rightarrow P$ over expressions and formulas. Thus formulas can be either true, false or not defined and expressions may evaluate to a value or be undefined. The operator $\delta(E)$ $(\delta(P))$ gives the, necessary and sufficient conditions such that $E(P)$ is defined. More particularly, the application of the operator $\delta(E)$ $(\delta(P))$ over E (P) holds in a state only if the expression E has a value in this state, otherwise it is false.

In the following we shall abstract from partiality of evaluation and we shall consider that our evaluation is a total function. Note that we could have used as is done , a three valued interpretation for formulas and expressions with an operator for well-definedness over predicates and expressions whose interpretation to true would mean that the predicate / expression is not undefined. We however, consider that this only will complicate the present formalization without bringing any original feature of the present thesis. This means that for instance, for predicates which contain field dereference we shall assume that the reference is not **null**, that arrays are always accessed inside their bounds etc.

lilian

The relation $\vDash$ that we define next, gives a meaning to the formulas from our assertion language P w.r.t. two state configurations - a current state and an initial state.

Definition 4.3.1 (Interpretation of predicates) *The interpretation $s \vDash P$ of a predicate P in a state configuration $s =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ w.r.t. an*

initial state $s_0 =< \mathrm{H}_0, 0, [\,], \mathrm{Reg}, 0 >$ *is defined inductively as follows:*

$s, s_0 \vDash$ **true** *is true in any state* s

$s, s_0 \vDash$ **false** *is false in any state* s

$s, s_0 \vDash \neg\ P$ *if and only if* $\neg\ s, s_0 \vDash P$

$s, s_0 \vDash P_1 \wedge P_2$ *if and only if* $s, s_0 \vDash P_1$ *and* $s, s_0 \vDash P_2$

$s, s_0 \vDash P_1 \vee P_2$ *if and only if* $s, s_0 \vDash P_1$ *or* $s, s_0 \vDash P_2$

$s, s_0 \vDash P_1 \Rightarrow P_2$ *if and only if* $s, s_0 \vDash P_1$ *then* $s, s_0 \vDash P_2$

$s, s_0 \vDash \forall x : T.P(x)$ *if and only if forall value* $\mathbf{v}$ *of type* T $s, s_0 \vDash P(\ \mathbf{v})$

$s, s_0 \vDash \exists x : T.P(x)$ *if and only if a value* $\mathbf{v}$ *of type* T *exists such that* $s, s_0 \vDash P(\ \mathbf{v})$

$s, s_0 \vDash E_1\ \mathcal{R}\ E_2$ *if and only if* $[\![E_1]\!]_{s_0,s}$ *rel*$(\mathcal{R})[\![E_2]\!]_{s_0,s}$ *is true*

$s, s_0 \vDash$ **instances**(`ref`), *where* `ref` $\in$ *RefVal if and only if* `ref` $\in \mathrm{H}_0$.`Loc`

Next, we define a function for expression evaluation $[\![*]\!]_{*,*}$ which evaluates expressions in a state has the following signature:

$$[\![*]\!]_{*,*} : E \rightarrow S \rightarrow S \rightarrow \mathit{Values} \cup \mathit{JType}$$

Note that the evaluation function is total and takes as arguments an expression of the assertion language presented in the previuos Section 4.1 and two states (see Section 2.5) and returns a value as defined in Section 2.4.

Definition 4.3.2 (Evaluation of expressions) *The evaluation in a state* $s =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *or* $s =< \mathrm{H}, \mathrm{Reg} >^{final}$ *Final of an expression* E *w.r.t. an initial state* $s_0 =< \mathrm{H}_0, 0, [\,], \mathrm{Reg}, 0 >$ *is denoted with* $[\![E]\!]_{s_0,s}$ *and is defined inductively over the grammar of expressions* E *as follows:*

$$[\![v]\!]_{s_0,s} = v$$
$$\textit{where } v \in \textbf{ int } \vee \ v \in \textit{RefVal}$$

$$[\![f(E)]\!]_{s_0,s} =$$
$$= \mathrm{H}(f)([\![E]\!]_{s_0,s})$$

$$[\![f[\oplus E_1 \rightarrow E_2](E_3)]\!]_{s_0,s} =$$
$$= \mathrm{H}[\oplus f \rightarrow f[\oplus [\![E_1]\!]_{s_0,s} \rightarrow [\![E_2]\!]_{s_0,s}]](f)([\![E_3]\!]_{s_0,s})$$

$$[\![\textbf{arrayAccess}(E_1, E_2)]\!]_{s_0,s} =$$
$$= \mathrm{H}([\![E_1]\!]_{s_0,s}, [\![E_2]\!]_{s_0,s})$$

$$[\![\mathrm{arrAccess}[\oplus(E_1, E_2) \rightarrow E_3](E_4, E_5)]\!]_{s_0,s} =$$
$$= \mathrm{H}[\oplus([\![E_1]\!]_{s_0,s}, [\![E_2]\!]_{s_0,s}) \rightarrow [\![E_3]\!]_{s_0,s}]$$
$$([\![E_4]\!]_{s_0,s}, [\![E_5]\!]_{s_0,s})$$

$$[\![\textbf{reg}(i)]\!]_{s_0,s} = \mathrm{Reg}(i)$$

$$[\![\backslash\textbf{old}(E)]\!]_{s_0,s} = [\![E]\!]_{s_0,s_0}$$

$$[\![E_1 \ op \ E_2]\!]_{s_0,s} = [\![E_1]\!]_{s_0,s} op [\![E_2]\!]_{s_0,s}$$

$$[\![\backslash\textbf{typeof}(E)]\!]_{s_0,s} =$$
$$\begin{cases} \textbf{int} & [\![E]\!]_{s_0,s} \in \textbf{int} \\ \mathrm{H.TypeOf}\ ([\![E]\!]_{s_0,s}) & else \end{cases}$$

$$[\![\backslash\textbf{elemtype}(E)]\!]_{s_0,s} =$$
$$\{ \quad \texttt{T} \quad if\ \mathrm{H.TypeOf}\ ([\![E]\!]_{s_0,s}) = \texttt{T[ ]}$$

$$[\![\backslash\textbf{TYPE}]\!]_{s_0,s} = \texttt{java.lang.Class}$$

The evaluation of stack expressions can be done only in intermediate state configurations $s =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *:*

$$[\![\textbf{cntr}]\!]_{s_0,s} = \mathrm{Cntr}$$

$$[\![\textbf{st}(E)]\!]_{s_0,s} = \mathrm{St}([\![E]\!]_{s_0,s})$$

The evaluation of the following expressions can be done only in a final state $s =< \mathrm{H}, \mathrm{Reg} >^{final}$ *Final:*

$$[\![\backslash\textbf{result}]\!]_{s_0,s} = \mathrm{Res} \quad \textit{where } s =< \mathrm{H}, \mathrm{Reg} >^{norm} \mathrm{Res}$$
$$[\![\backslash\textbf{EXC}]\!]_{s_0,s} = \mathrm{Exc} \quad \textit{where } s =< \mathrm{H}, \mathrm{Reg} >^{exc} \mathrm{Exc}$$

4.4 Extending method declarations with specification

In the following, we propose an extension of the method formalization given in Section 2.3. The extension takes into account the method specification. The extended method structure is given below:

$$\textbf{Method} = \left\{ \begin{array}{ll} \mathsf{Name} & : \textbf{MethodName} \\ \mathsf{retType} & : \mathit{JType} \\ \mathsf{args} & : (\mathit{name} * \mathit{JType})[] \\ \mathsf{nArgs} & : \mathit{nat} \\ \mathsf{body} & : \ \mathrm{I}[] \\ \mathsf{excHndlS} & : \textbf{ExcHandler}[] \\ \mathsf{exceptions} & : \textbf{Class}_{exc}[] \\ \mathsf{pre} & : P \\ \mathsf{modif} & : E[\,] \\ \mathsf{excPostSpec} & : \mathit{ExcType} \rightharpoonup P \\ \mathsf{normalPost} & : P \\ \mathsf{loopSpecS} & : \textbf{LoopSpec}[\,] \end{array} \right\}$$

Let's see the meaning of the new elements in the method data structure.

- `m.pre` gives the precondition of the method, i.e. the predicate that must hold whenever `m` is called
- `m.normalPost` is the postcondition of the method in case `m`terminates normally
- `m.modif` is also called the method frame condition. It is a list of expressions that the method may modify during its execution
- `m.excPostSpec` is a total function from exception types to formulas which returns the predicate `m.excPostSpec(Exc)` that must hold in the method's poststate if the method `m` terminates on an exception of type `Exc`. Note that this function is constructed from the **exsures** clause of a method introduced in Chapter 3, section 3.3. For instance, if method `m` has an exsures clause:

  $$\textbf{exsures}\ (\ \texttt{Exc})\ \textbf{reg}(1) = \textbf{null}$$

  then for every exception type `SExc` such that subtype(`SExc` ,`Exc`) the function the result of the function `m.excPostSpec` for `SExc` is `m.excPostSpec(SExc)` = $\textbf{reg}(1) = \textbf{null}$. If for an exception `Exc` there is not specified exsures clause then the function excPostSpec returns the default exceptional postcondition predicate *false* , i.e. `m.excPostSpec(Exc)` = *false*
- `m.loopSpecS` is an array of **LoopSpec** data structures which give the specifcication information for a particular loop in the bytecode

The contents of a **LoopSpec** data structure is given hereafter:

$$\mathbf{LoopSpec} = \left\{ \begin{array}{ll} \mathsf{pos} & : nat \\ \mathsf{invariant} & : P \\ \mathsf{modif} & : E[\,] \end{array} \right\}$$

define modifies locations in the grammar

For any method $\mathtt{m}$ for any k such that $0 \leq k < \mathtt{m}.\mathsf{loopSpecS}.length$

- the field $\mathtt{m}.\mathsf{loopSpecS}[k].\mathsf{pos}$ is a valid index in the body of $\mathtt{m}$: $0 \leq \mathtt{m}.\mathsf{loopSpecS}[k].\mathsf{pos} < \mathtt{m}.\mathsf{body}.length$ and is a loop entry instruction in the sense of Def.2.9.1
- $\mathtt{m}.\mathsf{loopSpecS}[k].\mathsf{invariant}$ is the predicate that must hold whenever the instruction $\mathtt{m}.\mathsf{body}[\mathtt{m}.\mathsf{loopSpecS}[k].\mathsf{pos}]$ is reached in the execution of the method $\mathtt{m}$
- $\mathtt{m}.\mathsf{loopSpecS}[k].\mathsf{modif}$ are the locations such that for any two states $state_1$, $state_2$ in which the instruction $\mathtt{m}.\mathsf{body}[\mathtt{m}.\mathsf{loopSpecS}[k].\mathsf{pos}]$ executes agree on local variables and the heap modulo the locations that are in the list modif. We denote the equality between $state_1$, $state_2$ modulo the modifies locations like this $state_1 =^{\mathsf{modif}} state_2$

Chapter 5

Verification condition generator for Java bytecode

This section describes a Hoare style verification condition generator for bytecode based on a weakest precondition predicate transformer function.

The vcGen is tailored to the bytecode language introduced in Section 2.8 and thus, it deals with stack manipulation, object creation and manipulation, field access and update, as well as exception throwing and handling.

Different ways of generating verification conditions exist. The verification condition generator presented propagates the weakest precondition and exploits the information about the modified locations by methods and loops. In Section 5.1, we discuss the existing approaches and motivate the choices done here.

Bytecode verification has become lately quite fashionable, thus several works exist on bytecode verification. Section 5.2 is an overview of the existing work in the domain.

Performing Hoare style logic verification over an unstructured program like bytecode programs has few particularities which verification of structured programs lacks. For example, loops on source level correspond to a syntactic structure in the source language and thus, identifying a loop in a source program is not difficult. However, this is not the case for unstructured programs. As we saw in the previous section 3, our approach consists in compiling source specification into bytecode specification. When compiling a loop invariant, we need to know where exactly in the bytecode the invariant must hold. Section **??** introduces the notion of a loop in an unstructured program.

As we stated earlier, our verification condition generator is based on a weakest precondition (wp) calculus. As we shall see in Section 5.3 a wp function for bytecode is similar to a wp function for source code. However, a logic tailored to stack based bytecode should take into account particular bytecode features as for example the operand stack.

5.1 Discussion

Before getting into more technical details, we would first like to outline the general features of our verification condition generator.

Our verification condition generator has the following features :

based on a weakest precondition predicate transformer The weakest precondition generates a precondition predicate starting from the end of the program with a specified postcondition and "goes " in a backward direction to the entry point instruction of the program.

There is an alternative for generating verification condition which works in a forward direction called a strongest postcondition predicate transformer. However, strongest postcondition tends to generate large formulas which is less practical than the more concise formulae generated by the weakest precondition calculus. Next, it generates existential quantification for every assignment expression in a program which are not easily treated by automatic theorem provers. For more detailed information on strongest postcondition calculus the reader may refer to [27].

???

works directly on bytecode Another possible approach is to generate verification conditions over a guarded command language program. A guarded command language is a small programming language with few program constructs but which are sufficiently expressive to encode a rich programming language. If a guarded command language is used in the verification procedure this would mean a stage where bytecode programs are transformed in guarded command language programs. Using guarded command language as an intermediate representation of programs is useful because the verification condition generator can interface and can be extended to interface easily several programming languages. Such an intermediate representation is used in the extended static checker ESC/java ([46]) and Spec# ([11]).

However, we consider that a guarded command language is not completely suitable for a PCC architecture. More particularly, we consider that proving the transformation algorithm from a programming language to a guarded command language could be not trivial and thus, we prefer to keep a verification condition generator which works directly on bytecode programs.

propagates verification conditions up to the program entry instruction This means that the underlying weakest precondition calculus will discharge the verification conditions only when it has reached the program entry point.

For this feature we also have an alternative solution. The latter consists in that verification conditions are discharged immediately when an assertion (e.g. loop invariant) is reached by the verifiction condition generator as is done in the seminal paper of Floyd [63] (see also the definition of the

verification condition in [12]). These verification conditions (in the case of a weakest precondition predicate transformer) state in case of a loop invariant that the loop invariant implies the postcondition of the loop if the loop condition is not true and that the invariant implies the weakest precondition of the loop body if the loop condition holds.

Although, generating in this way verification conditions is simpler than our approach it needs much stronger invariants in order that they get provable.In particular, the specification required for this alternative approach may increase the size of the program considerably which could not be always admissible, for instance if the program and its specification must be sent via the network. Another shortcoming is that writing or inferring these stronger invariants may be difficult.

relie avec la transformation Benjamin

deals only with functional properties We assume that programs are well-typed i.e. that programs have passed successfully the bytecode verification. By well-typed program we mean that every instruction in the program starts execution in a state where for instance, the method operand stack contains the right number and values of the right type and where the heap is well typed. Our verification condition generator does not generate such kind of type constraints over the bytecode instructions.

In fact, we could have designed a logic for checking also for well - typedness as is done in the work of Benton [15].

However, we consider that this problem is well-understood with type checking algorithms and we prefer to concentrate on the functional aspect of programs. We shall not enter here in the details of the bytecode verification algorithms because, as we have already seen in Section 2.1, the field has been profoundly studied and the curious reader may refer to the existing literature (e.g. [47]).

5.2 Related work

In the following, we review briefly the existing work related to program verification and more particularly program verification tailored to Java and Java bytecode programs.

gilles: what do you mean by giving a title to every section

Floyd is among the first to work on program verification using logic methods for unstructured program languages (see [63]). Following the Floyd's approach, T.Hoare gives a formal logic for program verification in [35] known today under the name Hoare logic. Dijkstra and Scholten [27] proposes then an efficient way for applying Hoare logic in program verification, in particular they propose two predicate transformer calculus and give their formal semantics.

As Java has been gaining popularity in industry since the nineties of the twentieth century, it also attracted the research interest. Thus the nineties upto nowadays give rise to several verification tools tailored to Java based on Hoare logic. Among the ones that gained most popularity are esc/java developed at Compaq [46], the Loop tool [41], Krakatoa, Jack [20] etc.

Few works have been dedicated to the definition of a bytecode logic. Among the earliest work in the field of bytecode verification is the thesis of C.Quigley [61] in which Hoare logic rules are given for a bytecode like language. This work is limited to a subset of the Java virtual machine instructions and does not treat for example method calls, neither exceptional termination. The logic is defined by searching a structure in the bytecode control flow graph, which gives an issue to complex and weak rules.

The work by Nick Benton [15] gives a typed logic for a bytecode language with stacks and jumps. The technique that he proposes checks at the same time types and specifications. The language is simple and supports basically stack and arithmetic operations. Finally, a proof of correctness w.r.t. an operational semantics is given.

Following the work of Nick Benton, Bannwart and Muller [10] give a Hoare logic rules for a bytecode language with objects and exceptions. A compiler from source proofs into bytecode proofs is also defined. As in our work, they assume that the bytecode has passed the bytecode verification certification. The bytecode logic aims to express functional properties. Invariants are inferred by fixpoint calculation. However, inferring invariants is not a decidable problem.

The Spec# ([11]) programming system developed at Microsoft proposes a static verification framework where the method and class contracts (pre, post conditions, exceptional postconditions, class invariants) are inserted in the intermediate code . Spec# is a superset of the C# programming language, with a built-in specification language, which proposes a verification framework (there is a choice to perform the checks either at runtime or statically). The static verification procedure involves translation of the contract specification into metadata which is attached to the intermediate code. The verification procedure [51] that is performed includes several stages of processing the bytecode program: elimination of irreducible loops, transformation into an acyclic control flow graph, translation of the bytecode into a guarded passive command language program. Despite that here in our implementation we also do a transformation in the graph into an acyclic program, we consider that in a mobile code scenario one should limit the number of program transformations for several reasons. First, we need a verification procedure as simple as possible, and second every transformation must be proven correct which is not always trivial.

5.3 Weakest precondition calculus

The weakest precondition predicate transformer function which for any instruction of the Java sequential fragment determines the predicate that must hold in the prestate of the instruction has the following signature:

$$wp : (int,\ \mathrm{I}) \longrightarrow \mathbf{Method} \longrightarrow P$$

The function *wp* takes two arguments : the second argument is the method `m` to which the instruction belongs and the first argument is the instruction (for instance putfield) along with its position in `m`.

Let us first see what is the desired meaning of *wp*. Particularly, we would like that the function *wp* returns a predicate $wp(pos\ ins, \mathtt{m})$ such that if it holds in the prestate of the method `m` and if the `m` terminates normally then the normal postcondition `m.`normalPost holds when `m` terminates execution, otherwise if `m` terminates on an exception `Exc` the exceptional postcondition `m.`excPostSpec`(Exc)` holds where the function excPostSpec was introduced in Section 4.4. Thus, the *wp* function takes into account both normal and exceptional program termination. Note however, that *wp* deals only with partial correctness, i.e. it does not guarantee program termination. In the next Chapter 6, we shall formally argument that *wp* has this semantics.

In the following, we will give an intuition to the way in which we have defined our verification condition generator. Consider the example in Fig. 5.1 which shows both the source code and the bytecode of a method which calculates the sum of all the natural numbers smaller or equal to the parameter k. The source and bytecode are annotated, the first one in JML and the latter in BML. However, the bytecode annotations are actually stored separately from the bytecode instructions as we have described in Section 4.4 but we have put them explicitely in the bytecode at the point where they must hold for the sake of clarity. Note that in what follows we will name the loop invariant I. We have also marked the instructions which are identified as loop start and end according to Def.2.9 in Chapter 2.8, Section 2.9.

It is worth first to note that because the bytecode is not structured we cannot define the weakest precondition in the same way in which a predicate transformer for structured languages is defined. We will rather define the predicate transformer for an instructions to depend on the instructions which may follow it during an execution:

$$wp(j) = \bigwedge_k C_k \Rightarrow S_k(wp(k)), where\ j \rightarrow k$$

C_k stands for some condition to be filled in order that after the execution of instruction at index j the instruction at index k is executed. In most of the cases the condition C_k is *true* except for conditional jumps or when the instruction j throws an exception. S_k stands for a function which might be the identity function or a function which applies substitutions over its argument.

Thus the weakest precondition for the instruction at index 12 (in the bytecode version) which is a conditional jump of the program will be:

$$\begin{aligned} wp(12) = \quad & \mathbf{st}(\mathbf{cntr} - 1) < \mathbf{st}(\mathbf{cntr}) \Rightarrow wp(13)[\mathbf{cntr} \backslash \mathbf{cntr} - 2] \\ & \wedge \\ & \mathbf{st}(\mathbf{cntr} - 1) >= \mathbf{st}(\mathbf{cntr}) \Rightarrow wp(5)[\mathbf{cntr} \backslash \mathbf{cntr} - 2], \\ & where\ 12 \rightarrow 13, 12 \rightarrow 5 \end{aligned}$$

Let us see now what we would expect about the result of the function *wp* when applied to the instructions that have as successor the loop entry instruction at index 10. For instance, we can look at the instruction at index 9 which is marked in the figure as the end of the loop. As we said earlier we have inlined

```
1  //@requires reg(1)>=0
2  0 const 0
3  1 store 2
4  2 const 0
5  3 store 3
6  4 goto 10
7  5 load 2
8  6 load 3
9  7 add
10 8 store 2
11 9 iinc 3 //LOOP END
12 //@loop_modifies reg(2),reg(3)
13 //@loop_invariant I :=
14 //@                reg(3)>=0 &&
15 //@                reg(3)<=reg(1)&&
16 //@                reg(2)==reg(3)*(reg(3)-1)/2
17 10 load 3 //LOOP ENTRY
18 11 load 1
19 12 if_icmplt 5
20 13 return
21 //@ensures \result==
22            reg(1)*(reg(1)+1)/2
```

```
1  //@requires k >= 0 ;
2  //@ensures \result == k*(k+1)/2;
3  public void m(int k){
4    int sum = 0;
5    //@loop_modifies sum, i;
6    //@loop_invariant i >= 0
7    && i <=k && sum == i*(i-1)/2;
8    for (int i = 0;i < k;i++){
9      sum = sum + i;
10   }
11 }
```

Figure 5.1: BYTECODE OF METHOD SUM AND ITS SPECIFICATION

annotations in the bytecode at the places where they must hold. Thus after the execution of the instruction at index 9 the loop invariant must hold. It follows then that for a loop end instruction we will rather require that the *wp* function takes into account the corresponding loop invariant:

$$wp(9) = \begin{array}{l} I[\mathbf{reg}(3)\backslash\mathbf{reg}(3)+1], \\ where\ 9 \rightarrow^{l} 10 \end{array}$$

The situation is similar for the instruction at index 4 which jumps to the loop entry instruction at index 10. The semantics of the invariant requires that in between instructions 4 and 10 first the invariant must hold and second, whatever are the values of the program variables that might be modified by the loop, the invariant should imply the precondition of the loop entry instruction

at 10. Thus we would like that the function *wp* gives us something like:

$$\begin{aligned} wp(4) = \quad & I \\ & \wedge \\ & \forall \mathbf{reg}(2), \mathbf{reg}(3), I \Rightarrow wp(10), \\ & \textit{where } 4 \rightarrow 10 \ \wedge 10 \textit{ is a loop entry} \end{aligned}$$

The example shows that the function *wp* depends on the semantics of the instruction for which it calculates a precondition and also on the execution relation it has with its successors. In order to define the function *wp* we will use an intermediate function which shall decide what is the postcondition of an instruction upon the execution relation with its successors. This function is introduced in the next subsection 5.3.1.

We will also see how the weakest precondition is defined in the presence of exceptions in subsection 5.3.2.

5.3.1 Intermediate predicates

In this subsection, we define the function *inter* which for two instructions that may execute one after another in a control flow graph of method `m` determines the predicate $inter(j, k, \mathtt{m})$ which must hold in between them. The function has the signature:

$$inter : int \longrightarrow int \longrightarrow \mathbf{Method} \longrightarrow P$$

The predicate $inter(j, k, \mathtt{m})$ will be used for determining the weakest predicate that must hold in the poststate of the instruction j in the execution path where j is followed by the instruction k. This predicate depends on the execution relation between the two instructions j and k. Recall that we introduced the notion of execution relation between two instructions in Chapter 2.8, Section 2.9.

Definition 5.3.1.1 (Intermediate predicate between two instructions) *Assume that $j \rightarrow k$. The predicate $inter(j, k, \mathtt{m})$ must hold after the execution of j and before the execution of k and is defined as follows:*

- *if k is a loop entry instruction, $j \rightarrow^l k$ and $\mathtt{m}.\mathsf{loopSpecS}[s].\mathsf{pos} = k$ then the corresponding loop invariant must hold:*

$$inter(j, k, \mathtt{m}) \equiv \mathtt{m}.\mathsf{loopSpecS}[s].\mathsf{invariant}$$

- *else if k is a loop entry and $\mathtt{m}.\mathsf{loopSpecS}[s].\mathsf{pos} = k$ then the corresponding loop invariant $\mathtt{m}.\mathsf{loopSpecS}[s].\mathsf{invariant}$ must hold before k is executed, i.e. after the execution of j. We also require that $\mathtt{m}.\mathsf{loopSpecS}[s].\mathsf{invariant}$ implies the weakest precondition of the loop entry instruction. The implication is quantified over the locations $\mathtt{m}.\mathsf{loopSpecS}[s].\mathsf{modif}$ that may be modified in the loop body:*

$$
\begin{array}{l}
inter(j,k,\mathtt{m}) \equiv \\
\mathtt{m}.\mathsf{loopSpecS}[s].\mathsf{invariant} \wedge \\
\forall i, i = 1..\mathtt{m}.\mathsf{loopSpecS}[s].\mathsf{modif}.length, \\
\forall \mathtt{m}.\mathsf{loopSpecS}[s].\mathsf{modif}[i], (\\
\qquad\qquad \mathtt{m}.\mathsf{loopSpecS}[s].\mathsf{invariant} \Rightarrow \\
\qquad\qquad\qquad wp(k\ , \mathtt{m}))
\end{array}
$$

- *else*

$$inter(j,k,\mathtt{m}) \equiv wp(k\ , \mathtt{m})$$

5.3.2 Weakest precondition in the presence of exceptions

Our weakest precondition calculus deals with exceptional termination and thus, we need a way for calculating the postcondition of an instruction in case it terminates on an exception. In particular, the postcondition should depend on if there is an exception handler or not. In the first case, the execution continues at the exception handler entry point and thus the postcondition of the exceptionally terminating instruction will be the precondition of the instruction from which the exception handler starts. In the case where there is no exception handler, this means that the current method also terminates on exception and thus, the specified exceptional exceptional postcondition of the method for this exception should hold.

We define the function excPostIns with signature :

$$\mathsf{excPostIns} : int \longrightarrow ExcType \longrightarrow P$$

The function m.excPostIns takes as arguments an index i in the array of instructions of method m and an exception type Exc and returns the predicate m.excPostIns(i, Exc) that must hold after the instruction at index i throws an exception of type Exc. We give a formal definition hereafter.

Definition 5.3.2.1 (Postcondition in case of a thrown exception)

$$
\mathtt{m}.\mathsf{excPostIns}(i,\ \mathtt{Exc}) = \\
\begin{cases}
inter(i, \mathsf{handlerPc}, \mathtt{m}) & if\ findExcHandler(\ \mathtt{Exc}, i, \mathtt{m}.\mathsf{excHndlS}) = \mathsf{handlerPc} \\
\mathtt{m}.\mathsf{excPostSpec}(\ \mathtt{Exc}) & findExcHandler(\ \mathtt{Exc}, i, \mathtt{m}.\mathsf{excHndlS}) = \bot
\end{cases}
$$

Next, we introduce an auxiliary function which will be used in the definition of the *wp* function for instructions that may throw runtime exceptions. Thus, for every method m we define the auxiliary function m.excPostRTE with signature:

$$\mathtt{m}.\mathsf{excPostRTE} : int \longrightarrow ExcType \longrightarrow P$$

m.excPostRTE(i, Exc) returns the predicate that must hold in the preststate of the instruction at index i which may throw a runtime exception of type Exc. Note that the function m.excPostRTE does not deal with programmatic

exceptions thrown by the instruction athrow, neither exception caused by a method invokation (execution of instruction invoke) as the exceptions thrown by those instructions are handled in a different way as we shall see later in the definition of the *wp* function in the next subsection.

The function application $\mathtt{m}.\mathsf{excPostRTE}(\ i,\ \mathtt{Exc})$ is defined as follows:

Definition 5.3.2.2 (Auxuliary function for instructions throwing runtime exceptions)

$$
\begin{array}{l}
i \neq \text{athrow} \wedge i \neq \text{invoke} \Rightarrow \\
\mathtt{m}.\mathsf{excPostRTE}(\ i,\ \mathtt{Exc}) = \\
\forall \mathtt{ref}, \\
\quad \neg\ \mathbf{instances}(\mathtt{ref}) \wedge \\
\quad \mathtt{ref} \neq \mathbf{null} \Rightarrow \\
\qquad \mathtt{m}.\mathsf{excPostIns}(\ i,\ \mathtt{Exc}) \\
\qquad [\mathbf{cntr} \backslash 0] \\
\qquad [\mathbf{st}(0) \backslash \mathtt{ref}] \\
\qquad [f \backslash f[\oplus \mathtt{ref} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f:\mathbf{Field},\ \mathsf{subtype}(f.\mathsf{declaredIn},\ \mathtt{Exc})} \\
\qquad [\backslash \mathbf{typeof}(\mathtt{ref}) \backslash\ \mathtt{Exc}]
\end{array}
$$

The function $\mathtt{m}.\mathsf{excPostRTE}$ will return a predicate which states that for every newly created exception reference the predicate returned by the function $\mathsf{excPostIns}$ must hold.

5.3.3 Rules for single instruction

In the following, we give the definition of the weakest precondition function for every instruction.

- Control transfer instructions

  1. unconditional jumps

     $$wp(i \text{ goto } n, \mathtt{m}) = \ inter(i, n, \mathtt{m})$$

     The rule says that an unconditional jump does not modify the program state and thus, the postcondition and the precondition of this instruction are the same

  2. conditional jumps

     $$
     \begin{array}{l}
     wp(i \text{ if\_cond } n, \mathtt{m}) = \\
     \quad \mathtt{cond}(\mathbf{st}(\mathbf{cntr}), \mathbf{st}(\mathbf{cntr} - 1)) \Rightarrow \\
     \qquad inter(i, n, \mathtt{m})[\mathbf{cntr} \backslash \mathbf{cntr} - 2] \\
     \wedge \\
     not(\ \ \mathtt{cond})(\mathbf{st}(\mathbf{cntr}), \mathbf{st}(\mathbf{cntr} - 1))) \Rightarrow \\
     \qquad inter(i, i + 1, \mathtt{m})[\mathbf{cntr} \backslash \mathbf{cntr} - 2]
     \end{array}
     $$

     In case of a conditional jump, the weakest precondition depends on if the condition of the jump is satisfied by the two stack top elements. If the condition of the instruction evaluates to true then

the predicate between the current instruction and the instruction at index n must hold where the stack counter is decremented with 2 $inter(i, n, \mathtt{m})[\mathbf{cntr}\backslash\mathbf{cntr} - 2]$ If the condition evaluates to false then the predicate between the current instruction and its next instruction holds where once again the stack counter is decremented with two $inter(i, i+1, \mathtt{m})[\mathbf{cntr}\backslash\mathbf{cntr} - 2]$.

3. return

$$wp(\mathtt{m}\ \text{return}, i) = \mathtt{m}.\mathsf{normalPost}[\backslash\mathbf{result}\backslash\mathbf{st}(\mathbf{cntr})]$$

As the instruction return marks the end of the execution path, we require that its postcondition is the normal method postcondition normalPost. Thus, the weakest precondition of the instruction is normalPost where the specification variable $\backslash\mathbf{result}$ is substituted with the stack top element.

- load and store instructions

1. load a local variable on the operand stack

$$\begin{array}{l} wp(i\ \text{load}\ \ \mathtt{j}, \mathtt{m}) = \\ inter(i, i+1, \mathtt{m})\ \begin{array}{l}[\mathbf{cntr}\backslash\mathbf{cntr} + 1] \\ [\mathbf{st}(\mathbf{cntr} + 1)\backslash\mathbf{reg}(j)]\end{array} \end{array}$$

The weakest precondition of the instruction then is the predicate that must hold between the current instruction and its successor, but where the stack counter is incremented and the stack top is substituted with $\mathbf{reg}(j)$. For instance, if we have that the predicate $inter(i, i+1, \mathtt{m})$ is equal to $\mathbf{st}(counter) == 3$ then we get that the precondition of instruction is $\mathbf{reg}(j) == 3$:

$$\begin{array}{l} \{\mathbf{reg}(j) == 3\} \\ i : \text{load}\ \ \mathtt{j} \\ \{\mathbf{st}(\mathbf{cntr}) == 3\} \\ i + 1 : \ldots \end{array}$$

2. store the stack top element in a local variable

$$\begin{array}{l} wp(i\ \text{store}\ \ \mathtt{j}, \mathtt{m}) = \\ inter(i, i+1, \mathtt{m})\ \begin{array}{l}[\mathbf{cntr}\backslash\mathbf{cntr} - 1] \\ [\mathbf{reg}(j)\backslash\mathbf{st}(\mathbf{cntr})]\end{array} \end{array}$$

Contrary to the previous instruction, the instruction store j will take the stack top element and will store its contents in the local variable $\mathbf{reg}(j)$.

3. push an integer constant on the operand stack

$$wp(i \text{ push } \mathtt{j}, \mathtt{m}) = \\ inter(i, i+1, \mathtt{m}) \begin{array}{l}[\mathbf{cntr} \backslash \mathbf{cntr} + 1] \\ [\mathbf{st}(\mathbf{cntr}+1) \backslash \mathtt{j}\ ]\end{array}$$

The predicate that holds after the instruction holds in the prestate of the instruction but where the stack counter **cntr** is incremented and the constant j is stored in the stack top element

4. incrementing a local variable

$$wp(\mathtt{m} \text{ iinc } \mathtt{j}, i) = \\ inter(i, i+1, \mathtt{m})[\mathbf{reg}(j) \backslash \mathbf{reg}(j) + 1]$$

- arithmetic instructions

1. instructions that cannot cause exception throwing (`arithOp` = add, sub, mult, and, or, xor , ishr, ishl,)

$$wp(i \text{ arith\_op}, \mathtt{m}) = \\ inter(i, i+1, \mathtt{m}) \begin{array}{l}[\mathbf{cntr} \backslash \mathbf{cntr} - 1] \\ [\mathbf{st}(\mathbf{cntr}-1) \backslash \mathbf{st}(\mathbf{cntr}) \mathtt{op}\ \mathbf{st}(\mathbf{cntr}-1)]\end{array}$$

We illustrate this rule with an example. Let us have the arithmetic instruction add at index i such that the predicate $inter(i, i+1, \mathtt{m}) \equiv \mathbf{st}(\mathbf{cntr}) \geq 0$. In this case, applying the rule we get that the weakest precondition is $\mathbf{st}(\mathbf{cntr}-1) + \mathbf{st}(\mathbf{cntr}) \geq 0$:

$$\{\mathbf{st}(\mathbf{cntr}-1) + \mathbf{st}(\mathbf{cntr}) \geq 0\} \\ i : \text{add} \\ \{\mathbf{st}(\mathbf{cntr}) \geq 0\}$$

2. instructions that may throw exceptions (`arithOp` = rem, div)

$$wp(i \text{ arithOp }, \mathtt{m}) = \\ \mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\ inter(i, i+1, \mathtt{m}) \begin{array}{l}[\mathbf{cntr} \backslash \mathbf{cntr} - 1] \\ [\mathbf{st}(\mathbf{cntr}-1) \backslash \mathbf{st}(\mathbf{cntr})\ \mathtt{op}\ \mathbf{st}(\mathbf{cntr}-1)]\end{array}$$

$$\wedge$$

$$\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathtt{m}.\mathsf{excPostRTE}(i,\ \mathtt{NullPntrExc})$$

- object creation and manipulation

1. create a new object

$$
\begin{array}{l}
wp(i \text{ new } \texttt{Cl}, \texttt{m}) = \\
\forall \mathbf{bv}, \\
\quad \neg\ \mathbf{instances}(\mathbf{bv}) \wedge \\
\quad \mathbf{bv} \neq \mathbf{null} \\
\quad \backslash\mathbf{typeof(bv)} <: \texttt{Cl} \Rightarrow \\
\qquad inter(i, i+1, \texttt{m}) \\
\qquad [\mathbf{cntr} \backslash \mathbf{cntr} + 1] \\
\qquad [\mathbf{st}(\mathbf{cntr}+1) \backslash \mathbf{bv}] \\
\qquad [f \backslash f[\oplus \mathbf{bv} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f : \mathbf{Field}.\mathsf{subtype}(f.\mathsf{declaredIn}, \texttt{Cl})}
\end{array}
$$

   The postcondition of the instruction new is the intermediate predicate $inter(i, i+1, \texttt{m})$. The weakest precondition of the instruction says that for any reference **bv** if **bv** is not an instance reference in the initial state of the execution of `m` then the precondition is the same predicate but in which the stack counter is incremented and **bv** is pushed on the stack top where the fields for the **bv** have the default value of their type.

2. array creation

$$
\begin{array}{l}
wp(i \text{ newarray } \texttt{T}, \texttt{m}) = \\
\forall \texttt{ref}, \\
\quad \textit{not } \mathbf{instances}(\texttt{ref}) \wedge \\
\quad \texttt{ref} \neq \mathbf{null} \wedge \\
\quad \backslash\mathbf{typeof(ref)} = \backslash\mathbf{type}(\texttt{T}[]) \wedge \\
\quad \mathbf{st}(\mathbf{cntr}) \geq 0 \Rightarrow \\
\qquad inter(i, i+1, \texttt{m}) \\
\qquad [\mathbf{st}(\mathbf{cntr}) \backslash \texttt{ref}] \\
\qquad [\text{arrAccess} \backslash \text{arrAccess}[\oplus(\texttt{ref}, j) \rightarrow \mathsf{defVal}(\texttt{T})]]_{\forall j, 0 \leq j < \mathbf{st}(\mathbf{cntr})} \\
\qquad [\text{arrLength} \backslash \text{arrLength}[\oplus \texttt{ref} \rightarrow \mathbf{st}(\mathbf{cntr})]] \\
\wedge \\
\mathbf{st}(\mathbf{cntr}) < 0 \Rightarrow \texttt{m}.\mathsf{excPostRTE}(i,\ \texttt{NegArrSizeExc})
\end{array}
$$

   Here, the rule for array creation is similar to the rule for object creation. However, creation of an array might terminate exceptionally in case the length of the array stored in the stack top element **st**(**cntr**) is smaller than 0. In this case, function `m`.excPostRTE will search for the corresponding postcondition of the instruction at position i and the exception `NegArrSizeExc`

3. field access

$$
\begin{array}{l}
wp(i \text{ getfield } f, \texttt{m}) = \\
\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\
\qquad inter(i, i+1, \texttt{m})[\mathbf{st}(\mathbf{cntr}) \leftarrow f(\mathbf{st}(\mathbf{cntr}))] \\
\wedge \\
\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \texttt{m}.\mathsf{excPostRTE}(i,\ \texttt{NullPntrExc})
\end{array}
$$

The instruction for accessing a field value takes as postcondition the predicate that must hold between it and its next instruction $inter(i, i+1, \mathtt{m})$. This instruction may terminate normally or on an exception. In case the stack top element is not **null**, the precondition of getfield is its postcondition where the stack top element is substituted by the field access expression $f(\mathbf{st}(\mathbf{cntr}))$. If the stack top element is **null**, then the instruction will terminate on a `NullPntrExc` exception. In this case the precondition of the instruction is the predicate returned by the function $\mathtt{m}.\mathsf{excPostRTE}$ for position i in the bytecode and exception `NullPntrExc`

4. field update

$$
\begin{array}{l}
wp(i \text{ putfield } f, \mathtt{m}) = \\
\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\
\quad inter(i, i+1, \mathtt{m}) \begin{array}{l}[\mathbf{cntr} \backslash \mathbf{cntr} - 2] \\ [f \backslash f[\oplus \mathbf{st}(\mathbf{cntr} - 1) \rightarrow \mathbf{st}(\mathbf{cntr})]]\end{array} \\
\wedge \\
\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathtt{m}.\mathsf{excPostRTE}(i,\ \mathtt{NullPntrExc})
\end{array}
$$

This instruction also may terminate normally or exceptionally. The termination depends on the value of the stack top element in the prestate of the instruction. If the top stack element is not **null** then in the precondition of the instruction $inter(i, i+1, \mathtt{m})$ must hold where the stack counter is decremented with two elements and the f object is substituted with an updated version $f[\oplus \mathbf{st}(\mathbf{cntr} - 2) \rightarrow \mathbf{st}(\mathbf{cntr} - 1)]$
For example, let us have the instruction putfield f in method `m`. Its normal postcondition is $inter(i, i+1, \mathtt{m}) \equiv f(\mathbf{reg}(1)) \neq \mathbf{null}$. Assume that `m` does not have exception handler for `NullPntrExc` exception for the region in which the putfield instruction. Let the exceptional postcondition of `m` for `NullPntrExc` be *false* , i.e. $\mathtt{m}.\mathsf{excPostSpec}($ `NullPntrExc`$)$ $=$ *false* If all these conditions hold, the function *wp* will return for the putfield instruction the following formula :

$$
\begin{array}{l}
\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\
\quad (f(\mathbf{reg}(1)) \neq \mathbf{null}) \begin{array}{l}[\mathbf{cntr} \backslash \mathbf{cntr} - 2] \\ [f \backslash f[\oplus \mathbf{st}(\mathbf{cntr} - 1) \rightarrow \mathbf{st}(\mathbf{cntr})]]\end{array} \\
\wedge \\
\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathit{false}
\end{array}
$$

After applying the substitution following the rules described in Section 4.2, we obtain that the precondition is

$$
\begin{array}{l}
\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\
\quad f[\oplus \mathbf{st}(\mathbf{cntr} - 1) \rightarrow \mathbf{st}(\mathbf{cntr})](\mathbf{reg}(1)) \neq \mathbf{null} \\
\wedge \\
\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathit{false}
\end{array}
$$

Finally, we give the instruction putfieldits postcondition and the respective weakest precondition:

$$
\begin{array}{l}
\quad \mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\
\{\ f[\oplus \mathbf{st}(\mathbf{cntr}-1) \rightarrow \mathbf{st}(\mathbf{cntr})](\mathbf{reg}(1)) \neq \mathbf{null}\} \\
\quad \wedge \\
\quad \mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathit{false}
\end{array}\ \}
$$

i : putfield f
$\{f(\mathbf{reg}(1)) \neq \mathbf{null}\}$
$i+1 : \ldots$

5. access the length of an array

$$
\begin{array}{l}
\mathit{wp}(i \text{ arraylength}, \mathtt{m}) = \\
\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\
\quad \mathit{inter}(i, i+1, \mathtt{m})[\mathbf{st}(\mathbf{cntr}) \backslash \text{arrLength}(\mathbf{st}(\mathbf{cntr}))] \\
\wedge \\
\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathtt{m.excPostRTE}(i,\ \mathtt{NullPntrExc})
\end{array}
$$

The semantics of arraylengthis that it takes the stack top element which must be an array reference and puts on the operand stack the length of the array referenced by this reference. This instruction may terminate either normally or exceptionally. The termination depends on if the stack top element is **null** or not. In case $\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null}$ the predicate $\mathit{inter}(i, i+1, \mathtt{m})$ must hold where the stack top element is substituted with its length. The case when a `NullPntrExc`is thrown is similar to the previous cases with exceptional termination

6. checkcast

$$
\begin{array}{l}
\mathit{wp}(i \text{ checkcast } C, \mathtt{m}) = \\
\backslash\mathbf{typeof}(\mathbf{st}(\mathbf{cntr})) <: C \vee \mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \\
\quad \mathit{inter}(i, i+1, \mathtt{m}) \\
\wedge \\
\mathit{not}(\backslash\mathbf{typeof}(\mathbf{st}(\mathbf{cntr})) <: C) \Rightarrow \mathtt{m.excPostRTE}(i, \mathtt{CastExc})
\end{array}
$$

The instruction checks if the stack top element can be cast to the class C. Two termination of the instruction are possible. If the stack top element $\mathbf{st}(\mathbf{cntr}\ )$ is of type which is a subtype of class C or is **null** then the predicate $\mathit{inter}(i, i+1, \mathtt{m})$ holds in the prestate. Otherwise, if $\mathbf{st}(\mathbf{cntr}\ )$ is not of type which is a subtype of class C, the instruction terminates on `CastExc` and the predicate returned by `m.excPostRTE` for the position i and exception `CastExc` must hold

7. instanceof

$$
\begin{array}{l}
\mathit{wp}(i \text{ instanceof } C, \mathtt{m}) = \\
\backslash\mathbf{typeof}(\mathbf{st}(\mathbf{cntr})) <: C \Rightarrow \\
\quad \mathit{inter}(i, i+1, \mathtt{m})[\mathbf{st}(\mathbf{cntr}) \backslash 1] \\
\wedge \\
\mathit{not}(\backslash\mathbf{typeof}(\mathbf{st}(\mathbf{cntr})) <: C) \vee \mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \\
\quad \mathit{inter}(i, i+1, \mathtt{m})[\mathbf{st}(\mathbf{cntr}) \backslash 0]
\end{array}
$$

This instruction, depending on if the stack top element can be cast to the class type C pushes on the stack top either 0 or 1. Thus, the rule is almost the same as the previous instruction checkcast.

- method invocation (only the case for non void instance method is given).

$$
\begin{array}{l}
wp(i \text{ invoke } \mathtt{n}\ , \mathtt{m}) = \\
\mathtt{n}\ .\mathsf{pre}[\mathbf{reg}(s) \leftarrow \mathbf{st}(\mathbf{cntr} + s - \mathtt{m}.\mathsf{nArgs})]_{s=0}^{\mathtt{n}\ .\mathsf{nArgs}} \\
\\
\wedge \\
\\
\forall mod, (mod \in \mathtt{n}\ .\mathsf{modif}), \forall freshVar(\\
\quad \mathtt{n}\ .\mathsf{normalPost} \begin{array}{l}[\backslash\mathbf{result}\backslash freshVar] \\ [\mathbf{reg}(s)\backslash\mathbf{st}(\mathbf{cntr} + s - \mathtt{n}\ ).\mathsf{nArgs}]_{s=0}^{\mathtt{n}\ .\mathsf{nArgs}}\end{array} \Rightarrow \\
\\
\qquad inter(i, i+1, \mathtt{m}) \begin{array}{l}[\mathbf{cntr}\backslash\mathbf{cntr} - \mathtt{n}\ .\mathsf{nArgs}] \\ [\mathbf{st}(\mathbf{cntr} - \mathtt{n}\ .\mathsf{nArgs})\backslash freshVar]\end{array}) \\
\\
\wedge_{j=0}^{\mathtt{n}\ .\mathsf{exceptions}.length-1} \\
\\
\forall mod, (mod \in \mathtt{n}\ .\mathsf{modif}), \\
\quad (findExcHandler(\mathtt{n}\ .\mathsf{exceptions}[j], i, \mathtt{m}.\mathsf{excHndlS}) = \bot \Rightarrow \\
\quad \forall \mathbf{bv}(\\
\qquad \mathtt{n}\ .\mathsf{excPostSpec}(\mathtt{n}\ .\mathsf{exceptions}[j])[\backslash\mathbf{EXC}\backslash\mathbf{bv}] \Rightarrow \\
\qquad\quad \mathtt{m}.\mathsf{excPostIns}(i, \mathtt{m}.\mathsf{exceptions}[j])[\backslash\mathbf{EXC}\backslash\mathbf{bv}])) \\
\quad \wedge \\
\quad (findExcHandler(\mathtt{m}\ .\mathsf{excPostSpec}(\mathtt{n}\ .\mathsf{exceptions}[j]), i, \mathtt{m}.\mathsf{excHndlS}) = k \Rightarrow \\
\quad \forall \mathbf{bv}(\\
\qquad \mathtt{n}\ .\mathsf{excPostSpec}(\mathtt{n}\ .\mathsf{exceptions}[j])[\backslash\mathbf{EXC}\backslash\mathbf{bv}] \Rightarrow \\
\qquad\quad inter(i, k, \mathtt{m}) \begin{array}{l}[\mathbf{cntr}\backslash 0] \\ [\mathbf{st}(0)\backslash\mathbf{bv}]\end{array}))
\end{array}
$$

Let us look in detail what is the meaning of the weakest precondition for method invokation. Because we are following a contract based approach the caller, i.e. the current method `m` must establish several facts. First, we require that the precondition `n.pre` of the invoked method `n` holds where the formal parameters are correctly initialized with the first `n.nArgs` elements from the operand stack.

Second, we get a logical statement which guarantees the correctness of the method invokation in case of normal termination. On the other hand, its postcondition `n.normalPost` is assumed to hold and thus, we want to establish that under the assumption that `m.normalPost` holds with **\result** substituted with a fresh bound variable **bv** and correctly initialized formal parameters is true we want to establish that the predicate *inter*(i, i+1 , `m`) holds . This implication is quantified over the locations `n.modif` that a

method may modify and the variable **bv** which stands for the result that the invoked method `n` returns.

The third part of the rule deals with the exceptional termination of the method invokation. In this case, if the invoked method `n` terminates on any exception which belongs to the array of exceptions `n.exceptions` that `n` may throw. Two cases are considered - either the thrown exception can be handled by `m` or not. If the thrown exception `Exc` can not be handled by the method `m` (i.e. $\mathit{findExcHandler}(\texttt{n .excPostSpec(n .exceptions}[j]), i, \texttt{m.excHndlS}) = \bot$) then if the exceptional postcondition predicate `n .excPostSpec(Exc )` of `n` holds then `m.excPostSpec(Exc )` for any value of the thrown exception object. In case the thrown exception`Exc` is handled by `m`, i.e. $\mathit{findExcHandler}(\texttt{n .excPostSpec(n .exceptions}[j]), i, \texttt{m.excHndlS}) = k$ then if the exceptional postcondition `n .excPostSpec(Exc )` of `n` holds then the intermediate predicate $\mathit{inter}(i, k, \texttt{m})$ that must hold after i and before k must hold once again for any value of thrown exception.

- throw exception instruction

$$
\begin{array}{l}
wp(i \text{ athrow }, \texttt{m}) = \\
\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \texttt{m.excPostIns}(i,\ \texttt{NullPntrExc}) \\
\wedge \\
\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \\
\ \forall \texttt{Exc}, \\
\ \backslash\mathbf{typeof(st(cntr))} <: \texttt{Exc} \Rightarrow \\
\quad \texttt{m.excPostIns}(i, \texttt{Exc})[\backslash\mathbf{EXC}\backslash\mathbf{st}(\mathbf{cntr})]
\end{array}
$$

The thrown object is on the top of the stack **st**(**cntr**). If the stack top object **st**(**cntr**) is **null**, then the instruction athrowwill terminate on an exception `NullPntrExc` where the predicate returned by the function `m.excPostRTE` must hold. The case when the thrown object is not **null** should consider all the possible exceptions that might be thrown by the current instruction. This is because we do not know the type of the thrown object which is on the stack top. The part of the *wp* when the thrown object on the stack top **st**(**cntr**) is not **null** considers all the possible types of the exception thrown. In any of

Supposing the execution of a method always terminates, the verification condition for a method `m` with a precondition `m.pre` is defined in the following way:

$$\texttt{m.pre} \Rightarrow wp(0\ \ \text{m .body}[0], \texttt{m})$$

5.4 Example

In the following, we shall see what are the resulting preconditions that the *wp* will calculate for the instructions in the bytecode from the program in Fig. 5.1.

Fig.5.2 shows the weakest preconditions for some of the instructions in the bytecode of the method sum. In the figure, the line before every instruction gives the calculated weakest precondition of the instruction. Thus, the weakest precondition of the instruction return at line 74 states that before the instruction is executed the stack top element **st**(**cntr**) must contain the sum of the natural numbers smaller than the local variable **reg**(1) . This precondition is calculated from the method postcondition which is given in curly brackets at line 75.

The instruction preceding the return instruction is a conditional branch which may jump to instruction at line 44 (or at position 5 in the bytecode array). This instruction has as precondition a predicate which reflects the two possible choices after it: if the element below the stack top **st**(**cntr** -1) is smaller than the stack top element **st**(**cntr**) then the precondition P5 of the instruction at line 44 must hold, otherwise the precondition Pre13 of the instruction at line 74 holds. For every instruction which does not targets a loop entry instruction the precondition is calculated from the precondition of its successor instructions. The special cases are the instructions at lines 37 and 56 which point to the loop entry instruction at line 61. As described earlier we can see that the resulting precondition of the instruction at line 56 is calculated upon the loop invariant. The precondition of the instruction at line 37 is calculated also upon the loop invariant but also confirms that the invariant implies the precondition of the loop entry instruction.

Finally, we can remark that the verification condition for the method Pre ==> Pre0 is valid.

```
1  {Pre: reg(1) >= 0}
2  {Pre0:=
3   0>=0 && 0<=reg(1)&& 0==0*(0-1)/2
4   &&
5   forall reg(3), reg(2)
6      reg(3)>=0 &&reg(3)<=reg(1)&&reg(2)==reg(3)*(reg(3)-1)/2
7         ==> reg(3)<reg(1) ==> Pre5
8         &&
9         reg(3)>= reg(1) ==>  Pre13}
10 0 const 0
11
12 {0>=0 && 0<=reg(1)&& st(cntr)==0*(0-1)/2
13  &&
14  forall reg(3), reg(2)
15     reg(3)>=0 &&reg(3)<=reg(1)&&reg(2)==reg(3)*(reg(3)-1)/2
16        ==> reg(3)<reg(1) ==> Pre5
17        &&
18        reg(3)>= reg(1) ==>  Pre13}
19 1 store 2
20
21 {0>=0 && 0<=reg(1)&& reg(2)==0*(0-1)/2
22  &&
23  forall reg(3), reg(2)
24     reg(3)>=0 &&reg(3)<=reg(1)&&reg(2)==reg(3)*(reg(3)-1)/2
25        ==> reg(3)<reg(1) ==> Pre5
26        &&
27        reg(3)>= reg(1) ==>  Pre13}
28 2 const 0
29
30 {st(cntr)>=0 && st(cntr)<=reg(1)&&reg(2)==st(cntr)*(st(cntr)+1)/2
31  &&
32  forall reg(3),reg(2),
33     reg(3)>=0&&reg(3)<=reg(1)&&reg(2)==reg(3)*(reg(3)-1)/2==>
34        reg(3)<reg(1) ==> Pre5  && reg(3)>= reg(1) ==> Pre13}
35 3 store 3
36
37 {I && forall reg(2),reg(3) (I ==> Pre10) }
38 4 goto 10
39
40 {Pre5:=
41  reg(3)+1>=0&&reg(3)+1<=reg(1)&&reg(2)+reg(3)==(reg(3)+1)*(reg(3))/2}
42 5 load 2
43
44 {reg(3)+1>=0&&reg(3)+1<=reg(1)&&st(cntr)+reg(3)==(reg(3)+1)*(reg(3))/2}
45 6 load 3
46
47 {reg(3)+1>=0&&reg(3)+1<=reg(1)&&
48  st(cntr-1)+st(cntr)==(reg(3)+1)*(reg(3))/2}
49 7 add
50
51 {reg(3)+1>=0&&reg(3)+1<=reg(1)&&
52  st(cntr)==(reg(3)+1)*(reg(3))/2}
53 8 store 2
54
55 {reg(3)+1>=0&&reg(3)+1<=reg(1)&&
56  reg(2)==(reg(3)+1)*(reg(3))/2}
57 9 iinc 3 //LOOP END
58
59 {Pre10 := reg(3)<reg(1) ==> Pre5
60           &&
61           reg(3)>= reg(1) ==>  Pre13}
62 10 load 3 //LOOP ENTRY
63
64 {st(cntr)<reg(1)==>Pre5
65  &&
66 st(cntr)>=reg(1)==>Pre13}
67 11 load 1
68
69 {st(cntr - 1) < st(cntr) ==> Pre5
70  &&
71  st(cntr - 1) >= st(cntr) ==>  Pre13}
72 12 if_icmplt 5
73
74 {Pre13:= st(cntr) ==reg(1)*(reg(1)+1)/2}
75 13 return
76 {Post: \result==reg(1)*(reg(1)+1)/2}
```

Figure 5.2: WEAKEST PRECONDITION PREDICATES FOR THE INSTRUCTIONS OF THE BYTECODE OF METHOD SUM

Chapter 6

Correctness of the verification condition generator

In the previous chapter, we defined a verification condition generator for a Java bytecode like language. We used a weakest precondition to build the verification conditions. In this section, we will show formally that the proposed verification condition generator is correct, or in other words that it is sufficient to prove the verification conditions generated over a method's body and its specification for establishing that the method respects the specification. In particular, we will prove the correctness of our methodology w.r.t. the operational semantics of our bytecode language given in chapter 2.8.

In the following, in Section 6.1 we first start with an outline of the proof. The second Section 6.2 establishes the relation between syntactic substitution and semantic evaluation. The latter will play a role in the correctness proof of the verification condition generator in Section 6.3.

6.1 Proof outline

We would first like to remark that in the following, we shall establish the soundness of our verification condition generator w.r.t. method correctness. For instance, the proof does not deal with the soundness of our methodology w.r.t. class invariants, class history constraints, neither about ghost variables. The reason for this is that their verification is the same on source and bytecode level. Note that establishing the semantics and verifying class invariants in JML in a modular way, which is our frontend language, is not completely clear. Class specifications is a subject which by itself is complex (see). Thus, we do not consider that these verification issues are particular to the bytecode verification techniques and thus, we shall omit to discuss their soundness here.

NB: pour les stacks est-ce qu'ils peuvent etre modifies? Le probleme c'est qu'on assume que le bytecode verifier a passe sur le programme, et donc pour chaque instruction le stack va avoir le meme type et profondeur avant l'execution de l'instruction. Alors est-ce que ca c'est grave?

ownership systems, Peter Mueller

Also, we assume that there are no recursive methods in the program. Proving the correctness of a verification calculus in the presence of recursive methods would require a notion of recursive method call depth in the operational semantics. This would complicate the proof without bringing any particular feature of the bytecode. The reader interested in how recursive methods are manipulated in the soundness proof of program logics might look at [57].

Remember that we assume that the control flow graph of a method is reducible, or in other words there are no cycles in the graph with two entries. This means that if program have cycles then they should conform to Def. 2.9.1 from Section 2.9. As we shall in the folloing, control flow graph reducibility plays an important role in the proof of soundness of the verification condition generator. Note that this restriction (as we said earlier) is realistic as every non-optimizing Java compiler produces reducible control flow graphs and even hand written code is usually reducible.

We would like to make a last but not least remark. The proof for soundness presented here would seem large w.r.t. the proof of soundness of Spec# presented in [51]. Spec# relies on several transformations over the bytecode. First, a transformation of the potentially irreducible control flow graph to a reducible one is done. The second step eliminates loops. The third step consists in translating programs into a single-assignment form. The fourth step is converting the program into a guarded command language and finally, the fifth and last step is passifying it which means changing assignments to `assume` statements. The proof presented in [51] is done for programs written in passified guarded command language. But what is the formal guarantee that the initial bytecode program is also correct? A proof of soundness for Spec# which takes into account only the last three stages can be already complex.

Before entering in the technical details of the proof, we give here an informal description of the steps to be overtaken there. We will describe our reasoning in the direction opposite to its formalization in the later sections. We start with our main objective and then start to "zoom" in the steps that must be made in order that it be achieved. Every time we discuss informally a step in the proof we point out the lemma in the formal proof to which it refers.

Let us first see what exactly we want to show. We would like to establish that if the verification conditions generated for a method are valid and the precondition of the method holds in the initial state of the method then the postcondition of the method holds in its final state. This is formally given in Definition 6.3.1, Section 6.3 and the main theorem 6.3.5 formally establishes that the verification conditions generated with the help of *wp* respect this definition.

To do this, we first show that if the precondition calculated by the *wp* for the entry instruction of the method holds in the initial state of the method then the postcondition of the method will hold in the final state of the method provided that the method terminates (Lemma 6.3.4).

For this, we have to establish that if the precondition calculated by the *wp* for the entry instruction of the method holds in the initial state of the method then the precondition calculated by *wp* for every instruction reachable from the method entry instruction also holds in the prestate of that instruction. This is

done in Lemma 6.3.3. We use here an induction over the number of execution steps made in the execution path. The induction case uses Lemma 6.3.2.

Lemma 6.3.2 shows that if in an execution path the preconditions calculated by *wp* for the instructions in the path are such that the respective precondition holds in the prestate of the respective instruction then either the respective normal or exceptional method postcondition holds or another execution step can be made and the weakest precondition of the next instruction holds. This is also known as progress property. The latter lemma is may be the most complicated one as it uses the argument of the reducibility of the execution graph (Section 2.9, Def. 2.9.1). The lemma has three cases:

- the case where the next instruction is not a loop entry instruction. In this case the proof is standard and uses the single step soundness of *wp*.
- the case when the next instruction is a loop entry instruction and the current instruction is not a loop end. In this case, we use the single step soundness of *wp* as well the special form of the precondition of the current instruction.
- the case where the next instruction is a loop entry and the current is a loop end instruction (see Def. 2.9.1). In this case, we use the reducibility of the control flow graph which gives us that the execution path has a prefix subpath which passes through the loop entry instruction but not through the current loop end instruction. This fact allows us to conclude that also in that case the Lemma 6.3.2 holds

What we mean by single step soundness of the *wp* function is that if the predicate calculated by *wp* for an instruction holds in its prestate then the postcondition upon which the precondition is calculated holds in its poststate (Lemma 6.3.1). The argument for the single step soundness uses the relation between syntactic substitution and semantic evaluation which looks like:

$$[\![E_1[E_2 \backslash E_3]]\!]_{s_0,s_1} = [\![E_1]\!]_{s_0,s_1[\oplus [\![E_2]\!]_{s_0,s_1} \rightarrow [\![E_3]\!]_{s_0,s_1}]}$$

This equivalence is standardly used for establishing the soundness of predicate transformer function w.r.t. a program semantics and means that it does not matter if we use a syntactic substitution over the expression and evaluate the resulting expression or update the state and evaluate the original expression. The next section is dedicated to the relation between substitution and evaluation.

6.2 Relation between syntactic substitution and semantic evaluation

In this section, we will show what is the relation between the syntactic notion of substitution and the semantic notion of evaluation. Particularly, we shall see

that they commute. As an intermediate execution state $< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ is composed from several elements, a heap H, the stack counter Cntr , the operand stack St and the array of registers Reg, we shall state for each component a separate lemma. In the following, we shall scatch the proof only of the first lemma the other lemmas having a similar proof.

Let us now look at the next formal statement. It refers to the fact that if we substitute in a give expression E_1 the expression $\mathbf{reg}(\mathrm{i})$ which represents the local variable at index i with another expression E_2 and evaluate the resulting expression in a state s_1 we will get the same value if we evaluate E_1 in the state $s_1[\oplus \mathrm{Reg}(i) \rightarrow [\![E_2]\!]_{s_0,s_1}]$.

Lemma 6.2.1 (Update a local variable) *For any expressions E_1, E_2 if we have that the states s_1 and s_2 are such that $s_1 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ and $s_2 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}[\oplus i \rightarrow [\![E_2]\!]_{s_0,s_1}], \mathrm{Pc} >$ then the following holds:*

1. $[\![E_1[\mathbf{reg}(i) \backslash E_2]]\!]_{s_0,s_1} = [\![E_1]\!]_{s_0,s_2}$
2. $s_1, s_0 \vDash \psi[\mathbf{reg}(i) \backslash E_2] \iff s_2, s_0 \vDash \psi$

Proof : by structural induction on the structure of E_1

1. we look at the first part of the lemma concerning expression evaluation

   - $E_1 = \mathbf{reg}(i)$

     $(left)\ \mathbf{reg}(i)[\mathbf{reg}(i) \backslash E_2] = E_2$
     $\Rightarrow$
     (1) $[\![\mathbf{reg}(i)[\mathbf{reg}(i) \backslash E_2]]\!]_{s_0,s_1} = [\![E_2]\!]_{s_0,s_1}$

     $(right)\ [\![\mathbf{reg}(i)]\!]_{s_0,s_2} =$
     { *by Def.4.3.2 of the evaluation for local variables* }
     (2) $= [\![E_2]\!]_{s_0,s_1}$
     { *from (1) and (2) we get that the lemma holds in this case* }

   - $E_1 = E_3.f$

     $E_3.f[\mathbf{reg}(i) \backslash E_2] =$
     { *by definition of the substitution* }
     $= E_3[\mathbf{reg}(i) \backslash E_2].f$
     { *by induction hypothesis* }
     (1)$[\![E_3[\mathbf{reg}(i) \backslash E_2]]\!]_{s_0,s_1} = [\![E_3]\!]_{s_0,s_2}$

     { *by Def.4.3.2 of the evaluation for field access expressions* }
     $(left)\ [\![E_3.f[\mathbf{reg}(i) \backslash E_2]]\!]_{s_0,s_1} =$
     $= \mathrm{H}(f)([\![E_3[\mathbf{reg}(i) \backslash E_2]]\!]_{s_0,s_1})$

     $(right)\ [\![E_3.f]\!]_{s_0,s_2} =$
     $= \mathrm{H}(f)([\![E_3]\!]_{s_0,s_2})$

     { *from (1),(left) and (right)*
     we get that the lemma holds in this case }

- the rest of the cases proceed in a similar way by appluing the induction hypothesis

2. second case of the lemma

- $\psi = E' \ \mathcal{R} \ E'$

$\{$ *from the first part of the lemma we get* $\}$
(1) $[\![E'[\mathbf{reg}(i)\backslash E_2]]\!]_{s_0,s_1} = [\![E']\!]_{s_0,s_2}$
(2) $[\![E''[\mathbf{reg}(i)\backslash E_2]]\!]_{s_0,s_1} = [\![E'']\!]_{s_0,s_2}$

$s_1, s_0 \vDash \psi[\mathbf{reg}(i)\backslash E_2]$
$\{$ *definition of substitution* $\}$
(3) $\equiv$
$s_1, s_0 \vDash E'[\mathbf{reg}(i)\backslash E_2] \ \mathcal{R} \ E''[\mathbf{reg}(i)\backslash E_2]$
$\{$ *by Def.4.3.1 we get* $\}$
$\Longleftrightarrow$
$[\![E'[\mathbf{reg}(i)\backslash E_2]]\!]_{s_0,s_1}$ *rel*$(\mathcal{R})[\![E''[\mathbf{reg}(i)\backslash E_2]]\!]_{s_0,s_1}$ *is true*
$\{$ *from (1) , (2) and (3)* $\}$
$\Longleftrightarrow$
$[\![E']\!]_{s_0,s_2}$ *rel*$(\mathcal{R})[\![E'']\!]_{s_0,s_2}$
$\equiv$
$s_2, s_0 \vDash \psi$

- the rest of the cases are by structural induction

Lemma 6.2.2 (Update of the heap) *For any expressions E_1, E_2, E_3 and any field f if we have that the states s_1 and s_2 are such that* $s_1 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *and*
$s_2 =< \mathrm{H}[\oplus f \rightarrow f[\oplus [\![E_2]\!]_{s_0,s_1} \rightarrow [\![E_3]\!]_{s_0,s_1}]], \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *the following holds*

1. $[\![E_1[f\backslash f[\oplus E_2 \rightarrow E_3]]]\!]_{s_0,s_1} = [\![E_1]\!]_{s_0,s_2}$
2. $s_1, s_0 \vDash \psi[f\backslash f[\oplus E_2 \rightarrow E_3]] \iff s_2, s_0 \vDash \psi$

Lemma 6.2.3 (Update of the heap with a newly allocated object) *For any expressions E_1 if we have that the states s_1 and s_2 are such that* $s_1 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ *and* $s_2 =< \mathrm{H}', \mathrm{Cntr}, \mathrm{St}[\oplus \mathrm{Cntr} \rightarrow [\![\mathtt{ref}]\!]_{s_0,s_1}], \mathrm{Reg}, \mathrm{Pc} >$ *where* $\mathrm{newRef}(\mathrm{H}, C) = (\mathrm{H}', \mathtt{ref})$ *the following holds*

1.

$$[\![E_1 \ {}^{[\mathbf{st}(\mathbf{cntr})\backslash \mathtt{ref}]}_{[f \leftarrow f[\oplus \mathtt{ref} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f:\mathbf{Field},\mathsf{subtype}(f.\mathsf{declaredIn},C)}}]\!]_{s_0,s_1}$$
$=$
$[\![E_1]\!]_{s_0,s_2}$

2.

$$s_1, s_0 \vDash \psi \; \begin{array}{l}[\mathbf{st}(\mathbf{cntr})\backslash \mathtt{ref}] \\ [f\backslash f[\oplus \mathtt{ref} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f:\mathbf{Field}, \mathsf{subtype}(f.\mathsf{declaredIn}, C)}\end{array}$$
$$\Longleftrightarrow$$
$$s_2, s_0 \vDash \psi$$

Lemma 6.2.4 (Update the stack) *For any expressions E_1, E_2, E_3 if we have that the states s_1 and s_2 are such that $s_1 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ and $s_2 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}[\oplus[\![E_2]\!]_{s_0,s_1} \rightarrow [\![E_3]\!]_{s_0,s_1}], \mathrm{Reg}, \mathrm{Pc} >$ then the following holds:*

1. $[\![E_1[\mathbf{st}(E_2)\backslash E_3]]\!]_{s_0,s_1} = [\![E_1]\!]_{s_0,s_2}$
2. $s_1, s_0 \vDash \psi[\mathbf{st}(E_2)\backslash E_3] \iff s_2, s_0 \vDash \psi$

Lemma 6.2.5 (Update the stack counter) *For any expressions E_1, E_2 if we have that the states s_1 and s_2 are such that $s_1 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ and $s_2 =< \mathrm{H}, [\![E_2]\!]_{s_0,s_1}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ then the following holds:*

1. $[\![E_1[\mathbf{cntr}\backslash E_2]]\!]_{s_0,s_1} = [\![E_1]\!]_{s_0,s_2}$
2. $s_1, s_0 \vDash \psi[\mathbf{cntr}\backslash E_2] \iff s_2, s_0 \vDash \psi$

Lemma 6.2.6 (Return value property) *For any expression E_1 and E_2, for any two states s_1 and s_2 such that $s_1 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >$ and $s_2 =< \mathrm{H}, [\![E_2]\!]_{s_0,s_1} >^{norm}$ then the following holds:*

1. $[\![E_1[\backslash\mathbf{result}\backslash E_2]]\!]_{s_0,s_1} = [\![E_1]\!]_{s_0,s_2}$
2. $s_1, s_0 \vDash \psi[\backslash\mathbf{result}\backslash E_2] \iff s_2, s_0 \vDash \psi$

The next definition defines a particular set of assertion formulas which we call valid formulas.

Definition 6.2.1 (Valid formulas) *If an assertion formula $f \in P$ holds in any current state and any initial state, i.e. $\forall s, s_0,\ s, s_0 \vDash f$ we say that this is a valid formula and we note it with : $\vDash f$*

6.3 Proof of Correctness

The correctness of our verification condition generator is established w.r.t. to the operational semantics described in Section 2.8. Remind that, we look only at partial correctness, i.e. our calculus works under the assumption that a progam terminates.

We first give a definition that a "method is correct w.r.t its specification"

Definition 6.3.1 (A method is correct w.r.t. its specification) *For every method* $\mathtt{m}$ *with precondition* $\mathtt{m}.\mathsf{pre}$*, normal postcondition* $\mathtt{m}.\mathsf{normalPost}$*and exceptional postcondition function* $\mathtt{m}.\mathsf{excPostSpec}$*, we say that* $\mathtt{m}$ *respects its specification if for every two states s_0 and s_1 such that :*

- $\mathtt{m} : s_0 \Rightarrow s_1$
- $s_0, s_0 \vDash \mathtt{m.pre}$

Then if m *terminates normally then the normal postcondition holds in the final state* s_1*:* $s_1, s_0 \vDash$ m.normalPost*. Otherwise, if* m *terminates on an exception* Exc *the exceptional postcondition holds in the poststate* s_1 $s_1, s_0 \vDash$ m.excPostSpec(Exc)

The next issue that is important for understanding our approach is that we follow the design by contract paradigm [16]. This means that when verifying a method body, we assume that the rest of the methods respect their specification in the sense of the previuos definition 6.3.1.

First, we establish the correctness of the weakest precondition function for a single instruction: if the *wp* (short for weakest precondition) of an instruction holds in the prestate then in the poststate of the instruction the postcondition upon which the wp is caclulated holds.

Lemma 6.3.1 (Single execution step correctness) *For every instruction* s*, for every state* $s_0 =< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, s >$ *and initial state* $s_0 =< \mathrm{H}_0, 0, [\,], \mathrm{Reg}, 0 >$ *of the execution of method* m *if the following conditions hold:*

- $\mathtt{m.body}[0] : s_0 \hookrightarrow^* s_n$
- $\mathtt{m.body}[s] : s_n \hookrightarrow s_{n+1}$
- $s_n, s_0 \vDash wp(\ \mathrm{Pc}_n, \mathtt{m})$
- $\forall\ \mathtt{n} : \mathbf{Method}.\ \mathtt{n} \neq \mathtt{m}$ *n is correct w.r.t. its specification*

then :

- *if* $\mathrm{Pc}_n \neq$ return *and the instruction does not terminate on exception,* $s_{n+1} =< \mathrm{H}_{n+1}, \mathrm{Cntr}_{n+1}, \mathrm{St}_{n+1}, \mathrm{Reg}_{n+1}, \mathrm{Pc}_{n+1} >$ *then* $s_{n+1}, s_0 \vDash$ *inter*$(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m})$ *holds*
- *if* $\mathrm{Pc}_n =$ return *then* $s_{n+1}, s_0 \vDash$ m.normalPost *holds*
- *else if* $\mathrm{Pc}_n \neq$ return *and the instruction terminates on a not handled exception* Exc *, then* $s_{n+1}, s_0 \vDash$ m.excPostSpec(Exc)

Proof : The proof is by case analysis on the type of instruction that will be next executed. We are going to see only the proofs for the instructions return, load, new and putfield the other cases being the same

1. $\mathrm{Pc}_n =$ return

{ *by initial hypothesis* }
$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash wp(\mathtt{m}\ \mathrm{return}, \mathrm{Pc}_n)$
{ *by definition the weakest precondition for* return }
$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash \mathtt{m}.\mathsf{normalPost}[\backslash\mathbf{result}\backslash\mathbf{st}(\mathrm{cntr})]$
{ *by the substitution property 6.2.6* }
$\Longleftrightarrow$
$< \mathrm{H}_n, [\![\mathbf{st}(\mathrm{cntr})]\!]_{s_0, <\mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n>} >^{norm}, s_0 \vDash \mathsf{normalPost}$
{ *by definition of the evaluation function* $[\![*]\!]_{*,*}$}
$\Longleftrightarrow$
$< \mathrm{H}_n, \mathrm{St}_n(\mathrm{Cntr}_n) >^{norm}, s_0 \vDash \mathsf{normalPost}$
{ *from the operational semantics for* return}
$s_{n+1} =< \mathrm{H}_n, \mathrm{St}_n(\mathrm{Cntr}_n) >^{norm}$
{ *and we obtain that* }
$s_{n+1}, s_0 \vDash \mathsf{normalPost}$

2. $\mathrm{Pc}_n = \mathrm{load}\ i$

Easy: say what are other cases are similar to this one

{ *by initial hypothesis* }
$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash wp(\mathrm{Pc}_n\ \mathrm{load}\ i, \mathtt{m})$
{ *definition of the wp function* }
$\equiv$
$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash \ inter(\mathrm{Pc}_n, \mathrm{Pc}_n + 1, \mathtt{m}) \begin{matrix}[\mathrm{cntr}\backslash\mathrm{cntr}+1] \\ [\mathbf{st}(\mathrm{cntr}+1)\backslash\mathbf{reg}(i)]\end{matrix}$
{ *applying the substitution properties 6.2.5 and 6.2.4* }
$\Longleftrightarrow$
(1) $< \mathrm{H}_n, \mathrm{Cntr}_n + 1, \mathrm{St}_n[\oplus\mathrm{Cntr}_n + 1 \rightarrow \mathrm{Reg}_n(i)], \mathrm{Reg}_n, \mathrm{Pc}_{n+1} >, s_0 \vDash$ $inter(\mathrm{Pc}_n, \mathrm{Pc}_n + 1, \mathtt{m})$
{ *from the operational semantics of the* load*instruction in section 2.8*}
(2) $s_{n+1} =< \mathrm{H}_n, \mathrm{Cntr}_n + 1, \mathrm{St}_n[\oplus\mathrm{Cntr}_n + 1 \rightarrow \mathrm{Reg}_n(i)], \mathrm{Reg}_n, \mathrm{Pc}_{n+1} >$
{ *from (1) and (2) follows* }
$s_{n+1}, s_0 \vDash \ inter(\mathrm{Pc}_n, \mathrm{Pc}_n + 1, \mathtt{m})$
{ *and the lemma holds in this case* }

3. $\mathrm{Pc}_n = \text{new } C$

$\{$ *by initial hypothesis* $\}$
$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash wp(\mathrm{Pc} \text{ new } C, \mathtt{m})$
$\{$ *definition of the wp function* $\}$
$\equiv$

$$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash \forall \mathtt{ref}, \textit{not } \mathbf{instances}(\mathtt{ref}) \wedge \mathtt{ref} \neq \mathbf{null} \Rightarrow$$

(1)
$$inter(i, i+1, \mathtt{m})\ [\mathbf{cntr} \backslash \mathbf{cntr} + 1]\ [\mathbf{st}(\mathbf{cntr}+1) \backslash \mathtt{ref}]\ [f \backslash f[\oplus \mathtt{ref} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f : \mathbf{Field}.\mathsf{subtype}(f.\mathsf{declaredIn}, C)}\ [\backslash \mathbf{typeof}(\mathtt{ref}) \backslash C]$$

$\{$ *from the operational semantics of* new *in section 2.8* $\}$

(2) $s_{n+1} =< \mathrm{H}_{n+1}, \mathrm{Cntr}_n + 1, \mathrm{St}_n[\oplus \mathrm{Cntr}_n + 1 \rightarrow \mathtt{ref}], \mathrm{Reg}_n, \mathrm{Pc}_n + 1 >$

(3) $\mathrm{newRef(H}, C) = (\mathrm{H}_{n+1}, \mathtt{ref}')$

$\{$ *following Def. 4.3.1 instantiate (1) with* $\mathtt{ref}'$ $\}$

$$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash \textit{not } \mathbf{instances}(\mathtt{ref}') \wedge \mathtt{ref}' \neq \mathbf{null} \Rightarrow$$

(4)
$$inter(i, i+1, \mathtt{m})\ [\mathbf{cntr} \backslash \mathbf{cntr} + 1]\ [\mathbf{st}(\mathbf{cntr}+1) \backslash \mathtt{ref}']\ [f \backslash f[\oplus \mathtt{ref}' \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f : \mathbf{Field}.\mathsf{subtype}(f.\mathsf{declaredIn}, C)}\ [\backslash \mathbf{typeof}(\mathtt{ref}) \backslash C]$$

$\{$ *from (3)* $\}$

(5) $< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash \textit{not } \mathbf{instances}(\mathtt{ref}') \wedge \mathtt{ref}' \neq \mathbf{null}$

$\{$ *from (4) and (5) and Def. 4.3.1* $\}$

$$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash inter(i, i+1, \mathtt{m})\ [\mathbf{cntr} \backslash \mathbf{cntr} + 1]\ [\mathbf{st}(\mathbf{cntr}+1) \backslash \mathtt{ref}']\ [f \backslash f[\oplus \mathtt{ref}' \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f : \mathbf{Field}.\mathsf{subtype}(f.\mathsf{declaredIn}, C)}\ [\backslash \mathbf{typeof}(\mathtt{ref}) \backslash C]$$

{*from lemmas 6.2.5, 6.2.4 and 6.2.2, 6.2.3 and the operational semantics of the instruction* new}
$s_{n+1}, s_0 \vDash inter(i, i+1, \mathtt{m})$

4. $\mathrm{Pc}_n = \text{putfield } f$

Easy: say what are other cases are similar to this one

$\{$ *by initial hypothesis* $\}$
$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash wp(\ \mathrm{Pc}_n$ putfield $f, \mathtt{m})$
$\{$ *definition of the wp function* $\}$
$\equiv$

$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash$
$\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow$
(1) $\quad inter(i, i+1, \mathtt{m})\ [\mathbf{cntr} \backslash \mathbf{cntr} - 2]\ [f \backslash f[\oplus \mathbf{st}(\mathbf{cntr} - 1) \rightarrow \mathbf{st}(\mathbf{cntr})]]$
$\wedge$
$\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathtt{m.excPostRTE}(i,\ \mathtt{NullPntrExc})$
$\{$ *we get three cases* $\}$

(a) the dereferenced reference on the stack top is **null** and an exception handler starting at instruction k exists for `NullPntrExc` and Pc_n is in its scope

$\{$ *thus, we get the hypothesis* $\}$
$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash \mathbf{st}(\mathbf{cntr}) = \mathbf{null}$
$\{$ *from the above conclusion and (1) we get* $\}$
$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash \mathtt{m.excPostRTE}(\mathrm{Pc}_n,\ \mathtt{NullPntrExc})$
$\{$ *from Def. 5.3.2.2 of the function* m.excPostRTE
??? and the assumption that the exception is handled we get $\}$
$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash$
$\forall \mathtt{ref},$
$\neg\ \mathbf{instances}(\mathtt{ref}) \wedge$
$\mathtt{ref} \neq \mathbf{null} \Rightarrow$
$inter(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m})$
$[\mathbf{cntr} \backslash 0]$
$[\mathbf{st}(0) \backslash \mathtt{ref}]$
$[f \backslash f[\oplus \mathtt{ref} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f:\mathbf{Field},\ \mathsf{subtype}(f.\mathsf{declaredIn},\ \mathtt{Exc})}$
$\{$ *from lemmas 6.2.5, 6.2.2, 6.2.4 and 6.2.3 and the operational semantics of* putfield$\}$
$s_{n+1}, s_0 \vDash\ inter(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m})$

(b) the reference on the stack top is **null** and and the exception thrown is not handled. In this case, we obtain following the same way of reasoning as the previous case :

$< \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >, s_0 \vDash$
$\forall \mathtt{ref},$
$\neg\ \mathbf{instances}(\mathtt{ref}) \wedge$
$\mathtt{ref} \neq \mathbf{null} \Rightarrow$
$\mathtt{m.excPostSpec}(\ \mathtt{NullPntrExc})$
$[\backslash \mathbf{EXC} \backslash \mathtt{ref}]$
$[f \backslash f[\oplus \mathtt{ref} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f:\mathbf{Field},\mathsf{subtype}(f.\mathsf{declaredIn},\ \mathtt{Exc})}$
$\{$ *from lemmas 6.2.4, 6.2.2, 6.2.3 and the operational semantics of* putfield$\}$
$s_{n+1}, s_0 \vDash \mathtt{m.excPostSpec}(\ \mathtt{NullPntrExc})$

(c) the reference on the stack top is not **null**

$$
\begin{array}{l}
\{ \ \textit{thus, we get the hypothesis} \ \} \\
< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >, s_0 \vDash \mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \\
\{ \ \textit{from the above conclusion and (1) we get} \ \} \\
< \mathrm{H}, \mathrm{Cntr}, \mathrm{St}, \mathrm{Reg}, \mathrm{Pc} >, s_0 \vDash \\
\quad \textit{inter}(i, i+1, \mathtt{m}) \ \begin{array}{l}[\mathbf{cntr} \backslash \mathbf{cntr} - 2] \\ [f \backslash f[\oplus \mathbf{st}(\mathbf{cntr} - 1) \rightarrow \mathbf{st}(\mathbf{cntr})]]\end{array} \\
\{ \ \textit{applying lemmas 6.2.5 and 6.2.2 and} \\
\ \textit{of the operational semantics of} \ \text{putfield}\} \\
s_{n+1}, s_0 \vDash \ \textit{inter}(i, i+1, \mathtt{m})
\end{array}
$$

We now establish a property of the correctness of the wp function for an execution path. The following lemma states that if the calculated preconditions of all the instructions in an execution path holds then either the execution terminates normally (executing a return) or exceptionally, or another step can be made and the *wp* of the next instruction holds.

Lemma 6.3.2 (Progress) *Assume we have a method* `m` *with normal postcondition* m.normalPost *and exception function* m.excPostSpec. *Assume that the execution starts in state*
$< \mathrm{H}_0, \mathrm{Cntr}_0, \mathrm{St}_0, \mathrm{Reg}_0, \mathrm{Pc}_0 >$ *and the there are made n execution steps causing the transitive state transition*
$< \mathrm{H}_0, \mathrm{Cntr}_0, \mathrm{St}_0, \mathrm{Reg}_0, \mathrm{Pc}_0 > \hookrightarrow^n < \mathrm{H}_n, \mathrm{Cntr}_n, \mathrm{St}_n, \mathrm{Reg}_n, \mathrm{Pc}_n >$. *Assume that* $\forall i, (\ 0 \leq i \leq n), \ s_i, s_0 \vDash \textit{wp}(\ \mathrm{Pc}_i, \mathtt{m})$ *holds then*

1. *if* $\mathrm{Pc}_n = \text{return}$ *then* $< \mathrm{H}_n, \mathrm{St}_n(\mathrm{Cntr}_n) >^{norm}, s_0 \vDash$ m.normalPost *holds.*

2. *if* $\mathrm{Pc}_n \neq \text{athrow}$ *throws a not handled exception of type* Exc $< \mathrm{H}_{n+1}, \mathtt{ref} >^{exc}, s_0 \vDash$ m.excPostSpec(Exc) *holds where* $\text{newRef}(\mathrm{H}_n, \mathrm{Exc}) = (\mathrm{H}_{n+1}, \mathtt{ref})$.

3. *if* $\mathrm{Pc}_n = \text{athrow}$ *throws a not handled exception of type* Exc $< \mathrm{H}_n, \mathrm{St}(\mathrm{Cntr}) >^{exc}, s_0 \vDash$ m.excPostSpec(Exc) *holds*

4. *else exists a state* s_{n+1} *such that another execution step can be done* $s_n \hookrightarrow s_{n+1}$ *and* $s_{n+1}, s_0 \vDash \textit{wp}(\ \mathrm{Pc}_{n+1}, \mathtt{m})$ *holds*

Proof : The proof is by case analysis on the type of instruction that will be next executed.

We consider three cases: the case when the next execution step doesnot enter a cycle (the next instruction is not a loop entry in the sense of Def.2.9.1) the case when the current instruction is a loop end and the next instruction to be executed is a loop entry instruction (the execution step is $\rightarrow_l$) and the case when the current instruction is not a loop end and the next instruction is a loop entry instruction (corresponds to the first iteration of a loop)

1. the next instruction to be executed is not a loop entry instruction.

   { *following Def. 5.3.1.1 of the function inter in this case* }
   (1) $inter(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m}) = wp(\ \mathrm{Pc}_{n+1}, \mathtt{m})$
   { *by initial hypothesis* }
   (2) $s_n, s_0 \vDash wp(\ \mathrm{Pc}_n, \mathtt{m})$
   { *from the previous lemma 6.3.1 and (2) , we know that* }
   (3) $s_{n+1}, s_0 \vDash\ inter(\mathrm{Pc}_n, \mathrm{Pc}_{n+1}, \mathtt{m})$
   { *from (1) and (3)* }
   $s_{n+1}, s_0 \vDash wp(\ \mathrm{Pc}_{n+1}, \mathtt{m})$

2. Pc_n is not a loop end and the next instruction to be executed is a loop entry instruction at index *loopEntry* in the array of bytecode instructions of the method $\mathtt{m}$(i.e. the execution step is of kind $\rightarrow^l$, see Def.2.9.1). Thus, there exists a natural number $i, 0 \leq i < \mathtt{m.loopSpecS}.length$ such that $\mathtt{m.loopSpecS}[i].\mathsf{pos} = loopEntry$, $\mathtt{m.loopSpecS}[i].\mathsf{invariant} = I$ and $\mathtt{m.loopSpecS}[i].\mathsf{modif} = \{mod_i, i = 1..s\}$. We look only at the case when the current instruction is a load instruction

$\{$ *by initial hypothesis* $\}$
$s_n, s_0 \vDash wp(\ \mathrm{Pc}_n \text{ load } i, \mathtt{m})$
$\{$ *by defintion of the wp function in section 5.3 of the previous chapter* $\}$
$s_n, s_0 \vDash \ inter(\mathrm{Pc}_n, \mathrm{Pc}_n + 1, \mathtt{m})\ \begin{matrix}[\mathbf{cntr}\backslash\mathbf{cntr}+1] \\ [\mathbf{st}(\mathbf{cntr}+1)\backslash\mathbf{reg}(j)]\end{matrix}$
$\{$*by the definition 5.3.1.1 for the case when*
the execution step is not a backedge but the target instruction is a loop entry $\}$
$s_n, s_0 \vDash$
$I\ \begin{matrix}[\mathbf{cntr}\backslash\mathbf{cntr}+1] \\ [\mathbf{st}(\mathbf{cntr}+1)\backslash\mathbf{reg}(i)]\end{matrix}$
$\wedge$
$\forall mod_i, i = 1..s(I \Rightarrow wp(\ \mathrm{Pc}_{n+1}, \mathtt{m}))\ \begin{matrix}[\mathbf{cntr}\backslash\mathbf{cntr}+1] \\ [\mathbf{st}(\mathbf{cntr}+1)\backslash\mathbf{reg}(i)]\end{matrix}$
$\{$ *from lemmas 6.2.5 and 6.2.4* $\}$
$\Longleftrightarrow$
$s_n\ \begin{matrix}[\mathrm{Cntr}\backslash[\![\mathbf{cntr}+1]\!]_{s_0,s_n}] \\ [\mathrm{St}\backslash\mathrm{St}[\oplus([\![\mathbf{cntr}+1]\!]_{s_0,s_n}) \rightarrow [\![\mathbf{reg}(i)]\!]_{s_0,s_n}]]\end{matrix}\ , s_0 \vDash$
$I \wedge$
$\forall mod_i, i = 1..s(I \Rightarrow wp(\ \mathrm{Pc}_{n+1}, \mathtt{m}))$
$\{$ *from the Def. 4.3 of the evaluation function* $\}$
$\equiv$
$s_n\ \begin{matrix}[\mathrm{Cntr}\backslash\mathrm{Cntr}+1] \\ [\mathrm{St}\backslash\mathrm{St}[\oplus\mathrm{Cntr}+1 \rightarrow \mathrm{Reg}(i)]]\end{matrix}\ , s_0 \vDash$
$I \wedge$
$\forall mod_i, i = 1..s(I \Rightarrow wp(\ \mathrm{Pc}_{n+1}, \mathtt{m}))$
$\{$ *from the operational semantics of* load $\}$
$s_{n+1}, s_0 \vDash \begin{matrix}I \wedge \\ \forall mod_i, i = 1..s(I \Rightarrow wp(\ \mathrm{Pc}_{n+1}, \mathtt{m}))\end{matrix}$
$\{$ *we can get from the last formulation* $\}$
(1) $s_{n+1}, s_0 \vDash I$

(2)$s_{n+1}, s_0 \vDash I \Rightarrow wp(\ \mathrm{Pc}_{n+1}, \mathtt{m})$
$\{$ *from (1) and (2)* $\}$
$s_{n+1}, s_0 \vDash wp(\ \mathrm{Pc}_{n+1}, \mathtt{m})$

3. Pc_n is an end of a cycle and the next instruction to be executed is a loop entry instruction at index *loopEntry* in the array of bytecode instructions of the method $\mathtt{m}$(i.e. the execution step is of kind $\rightarrow^l$). Thus, there exists a natural number $i, 0 \leq i < \mathtt{m.loopSpecS}.length$ such that $\mathtt{m.loopSpecS}[i].\mathsf{pos} = loopEntry$, $\mathtt{m.loopSpecS}[i].\mathsf{invariant} = I$ and $\mathtt{m.loopSpecS}[i].\mathsf{modif} = \{mod_i, i = 1..s\}$. We consider the case when the current instruction is a sequential instruction. The cases when the current instruction is a jump instruction are similar.

$\{$ *by hypothesis we get* $\}$

$$s_n, s_0 \vDash wp(\ \mathrm{Pc}_n, \mathtt{m})$$

{ *from Def. 5.3.1.1 and transformation over the above statement* }

$$(1) \quad s_{n+1}, s_0 \vDash I$$

{ *by hypothesis, loopEntry* $= \mathrm{Pc}_{n+1}$. *From def. 2.9.1, we conclude that there is a prefix subP* $= \mathtt{m}.\mathsf{body}[0] \rightarrow^*$ *loopEntry of the current execution path which does not pass through* Pc_n. *We can conclude that the transition between loopEntry and its predecessor k (which is at index k in* $\mathtt{m}.\mathsf{body}$*) in the path subP is not a backedge. By hypothesis we know that* $\forall i, 0 \leq i \leq n, s_i, s_0 \vDash wp(\ \mathrm{Pc}_i, \mathtt{m})$. *From def.5.3.1.1 and lemma 6.3.1 we conclude* }

$$\exists k, 0 \leq k \leq n \Rightarrow$$

$$(2\ ) \qquad \begin{array}{l} s_k, s_0 \vDash \\ I \\ \wedge \forall mod_i, i = 1..s(\ \begin{array}{l} I \Rightarrow \\ wp(\ loopEntry, \mathtt{m}) \end{array}\ ) \end{array}$$

$$(3) \qquad s_k =^{\mathsf{modif}} s_{n+1}$$

{ *because* $\mathtt{m}.\mathsf{loopSpecS}[i].\mathsf{modif} = \{mod_i, i = 1..s\}$ *and from (2) and (3)* }

$$(4)\ s_{n+1}, s_0 \vDash I \Rightarrow wp(\ loopEntry, \mathtt{m})$$

{ *from (1) and (4)* }

$$\begin{array}{l} s_{n+1}, s_0 \vDash wp(\ loopEntry, \mathtt{m}) \\ \Longleftrightarrow \\ s_{n+1}, s_0 \vDash wp(\ \mathrm{Pc}_{n+1}, \mathtt{m}) \end{array}$$

Qed.

We now shall see that starting execution of a method in a state where the *wp* predicate for the entry instruction holds implies that the *wp* for all the instructions in the execution path hold.

Lemma 6.3.3 (*wp* precondition for method entry point holds initially) *Assume we have a method* $\mathtt{m}$. *Assume that execution of method* $\mathtt{m}$ *starts execution in state* s_0 *and* $s_0, s_0 \vDash wp(\ \mathtt{m}.\mathsf{body}[0], \mathtt{m})$ *where and makes n steps to reach state* s_n*:* $s_0 \hookrightarrow^n s_n$, *then*

$$\forall i, 0 < i \leq n,\ s_i, s_0 \vDash wp(\ \mathtt{m}.\mathsf{body}[\mathrm{Pc}_i], \mathtt{m})$$

Proof : Induction over the number of execution steps n

1. $s_0 \hookrightarrow s_1$. *By initial hypothesis we have that* $s_0, s_0 \vDash wp(\ \mathtt{m.body}[0], \mathtt{m})$ *we can apply lemma 6.3.2, we get that* $s_1, s_0 \vDash wp(\ \mathrm{Pc}_1, \mathtt{m})$ *and thus, the case when one step is made from the initial state* s_0 *holds*

2. *Induction hypothesis:* $s_0 \hookrightarrow^{n-1} s_{n-1}$ *and* $\forall i, 0 < i \leq n-1,\ s_i, s_0 \vDash wp(\ \mathtt{m.body}[\mathrm{Pc}_i], \mathtt{m})$ *and there can be made one step* $s_{n-1} \hookrightarrow s_n$. *Lemma 6.3.2 can be applied and we get that (1)* $s_n, s_0 \vDash wp(\ \mathtt{m.body}[\mathrm{Pc}_n], \mathtt{m})$. *From the induction hypothesis and (1) follows that*

$$\forall i, 0 < i \leq n,\ s_i, s_0 \vDash wp(\ \mathtt{m.body}[\mathrm{Pc}_i], \mathtt{m})$$

Qed.

Having the last lemma we can establish that if a method starts execution in a state in which the *wp* precondition for the method entry instruction holds and the method terminates then the method postcondition holds in the method final state.

Lemma 6.3.4 (*wp* precondition for method entry point holds initially) *Assume we have a method* `m` *with normal postcondition* `m.normalPost` *and exception function* `m.excPostSpec`.

Assume that execution of method `m` *starts in state* $_0$ *and* $s_0, s_0 \vDash wp(0\ \mathtt{m.body}[0], \mathtt{m})$ *Then if the method* `m` *terminates, i.e. there exists a state* s_n, $\mathtt{m} : s_0 \Rightarrow s_n$ *such that* $\mathrm{Pc}_n = \mathrm{return}$ *or* Pc_n *throws an unhandled exception of type* `Exc` *the following holds:*

- *if* $\mathrm{Pc}_n = \mathrm{return}$ *then* $s_{n+1}, s_0 \vDash \mathtt{m.normalPost}$
- *if* Pc_n *throws a not handled exception of type* `Exc` *then* $s_{n+1}, s_0 \vDash \mathtt{m.excPostSpec(Exc)}$

Proof: *Let* $s_0 \hookrightarrow^* s_n$ *and* $\mathtt{m.body}[\mathrm{Pc}_n]$ *is a* return *or an instruction that throws a not handled exception. Applying lemma 6.3.3, we can get that* $\forall i, 0 \leq i \leq n,\ s_i, s_0 \vDash wp(\ \mathtt{m.body}[\mathrm{Pc}_i], \mathtt{m})$. *We apply lemma 6.3.2 for the case for a* return *or instruction that throws an unhandled exception which allows to conclude that the current statement holds. Qed.*

Now, we are ready to state the theorem which expresses the correctness of our verification condition generator w.r.t. the operational semantics of our language

Theorem 6.3.5 *For any method* `m` *if the verification condition is valid:*

$$(1)\ \vDash \mathtt{m.pre} \Rightarrow wp(\ \mathtt{m}\ .\mathtt{body}[0], \mathtt{m})$$

then `m` *is correct in the sense of the Def. 6.3.1.*

Proof: *As we want to prove that method* m *is correct w.r.t. its specification we have as hypothesis that the method precondition holds in the the initial state* s_0 *of the method execution, or* $s_0, s_0 \vDash \mathtt{m.pre}$. *From the latter and (1), we get that the weakest precondition of the entry point holds in the initial state, i.e.* $s_0, s_0 \vDash wp(\ \mathtt{m.body}[0], \mathtt{m})$. *We can apply then Lemma 6.3.4 and we get that the respective method postcondition holds in the final state of the method* m. *We can conclude then that* m *respects its specification in the sense of Def. 6.3.1.*
Qed.

Chapter 7

Equivalence between Java source and bytecode proof Obligations

In this chapter, we will look at the relationship between the verification conditions generated for a Java like source programming language and the verification conditions generated for the bytecode language as defined in Chapter 5. More particularly, we argue that they are syntactically equivalent if the compiler is nonoptmizing and satisfies certain conditions.

First, we would like to give the context and motivations for studying the relationship between source and bytecode verification.

Security becomes an important issue for the overall software quality. This is especially the case for mobile code or what so ever untrusted code. A solution proposed by G.Necula (see his thesis [55]) is the PCC framework which brings the possibility to a code receiver to verify if an unknown code or untrusted code respects certain safety conditions before being executed by the code receiver. In this framework the code client annotates the code automatically, generates verification conditons automatically and proves them automatically. The program accompanied by the proof is sent to the code receiver who will typecheck the proof against the verification conditions that he will generate. Because of its full automation this architecture fails to deal with complex functional or security properties.

The relation of the verification conditions over bytecode and source code can be used for building an alternative PCC architecture which can deal with complex security policy. More particularly, such an equivalence can be exploited by the code producer to generate the certificate interactively over the source code. Because of the equivalence between the proof obligations over source and bytecode programs (modulo names and types), their proof certificates are also the same and thus the certificate generated interactively over the source code can be sent along with the code.

In the following, Section 7.1 presents an overview of existing work in the field.

In Section 7.2, we introduce the source programming language. As we shall see, this programming language has the basic features of Java as it supports object creation and manipulation, like instance field access and update, exception throwing and handling, method calls as well as subroutines.

Section 7.3 presents a simple non optimizing compiler from the source language to the bytecode language presented already in Chapter **??**. In this section, we will discuss certain properties of the compiler which are necessary conditions for establishing this equivalence.

Next, Section 7.4 presents the weakest precondition predicate transformer which we define over the source language.

Section 7.5 introduces a new formulation of the weakest precondition for bytecode, which is defined over the compilation of source expressions and statements. This formulation of the weakest precondition is helpful for establishing the desired relation between bytecode and source proof obligations. In this section, we also discuss upon what conditions the weakest precondition given before in Chapter 5 and this new version are equivalent.

In Section 7.6, we proceed with the proof of equivalence between the proof obligations generated by the weakest precondition defined in Chapter 5. To do this, we first establish the equivalence between the source weakest precondition and the bytecode weakest precondition defined over the compilation of statements and expressions. We exploit this equivalence to conclude that the source verification condition generator and the bytecode verification condition generator presented in Chapter 5 are syntactically the same.

7.1 Related work

Several works dealing with the relation between the verification conditions over source and its compilation into a low level programming language exist.

Barthe, Rezk and Saabas in [12] also argue that proof obligations produced over source code and bytecode produced by a nonoptimizing compiler are equivalent. The source language which they use supports method invokation, exception throwing and handling. They do not consider instructions that may throw runtime exceptions which simplifies the reasoning over expressions. However, in their work they do not discuss what are the assumptions about the compiler or the properties that it might verify which will guarantee this equivalence. We claim that the proof presented in [12] for the verification condition preserving compilation is correct only if the non optimizing compiler has the properties discussed here in Section 7.3, subsection 7.3.4.

In [64], Saabas and Uustalu present a goto language provided with a compositional structure called SGoto. They give a Hoare logic rules for the language where they mention explicitely the program counter in the assertions and define a nonoptimizing compiler from the source language into SGoto. Few restrictions are imposed on the compiler. They show that if a source program has a

Hoare logic derivation against a pre and postcondition then its compilation in SGoto will also have a Hoare logic derivation in the aformentioned Hoare logic rules. However, we consider that this is not sufficient for a practical use especially when talking about programs written in an unstructured language. For instance, using this approach in a PCC scenario would mean that the consumer should know the structure of of the executable code in order that the proof can be checked. This may not be desirable for several reasons: the producer does not want to give the structure because he does not want that the producer may get the source version of its program; or even if he does it, this means that the verification process on the consumer side could become heavier as the unstructured program should be transformed somehow in a SGoto program. Thus, it is more relevant if possible to get directly a relation between a structured program and its compilation in an unstructured language which is the aim of the current chapter. For this, as Saabas and Uustalo we use a structured version of the compilation of a source program but which is an intermediate stage of our proof that source structured programs and bytecode unstructured programs have the same verifictaion conditions.

In [9], F.Bannwart and P.Muller show how to transform a Hoare style logic derivation on source Java like program into a Hoare style logic derivation of a Java bytecode like program. This solution however has the shortcoming that the certificate (in this case it is the Hoare style logic derivation) can be potentially large. In particular, this means that applying this technique in scenarios like PCC could be hardly applicable because of the large size of the certificate.

7.2 Source language

We present a source Java-like programming language which supports the following features: object manipulation and creation, method invokation, throwing and handling exceptions, subroutines etc. Fig. 7.1 gives the formal grammar of the source language.

As we can see from the figure, the language supports integer constants **constInt**, the boolean constants *true* and *false*, the null constant **null** denoting the empty reference, a construct **this** for referring to the current object, arithmetic expressions $\mathcal{E}^{src}$ arith_op $\mathcal{E}^{src}$ where arith_op $\in \{+, -, div, rem, *\}$. The language allows to talk about the value stored in a field f for the object reference $\mathcal{E}^{src}$ via the construct $\mathcal{E}^{src}.f$, cast expressions $(Class)\mathcal{E}^{src}$ and method local variables and parameters **var**. The language also has constructs for expressing method invokations $\mathcal{E}^{src}.m()$ as well as instance creation **new** $Class(\mathcal{E}^{src})$. For the sake of clarity we consider only non void methods which do not receive parameters and constructors that receive exactly one argument without losing any specific feature of modular object oriented languages. The language allows to express a relation between expressions via the construct $\mathcal{E}^{src}\ \mathcal{R}\ \mathcal{E}^{src}$ where $\mathcal{R} \in \{\leq, <, \geq, >, =, \neq\}$ as well as to state that an expression $\mathcal{E}^{src}$ is an instance of class Class via $\mathcal{E}^{src}$ **instanceof** $Class$.

The expressions can be of object types or basic types. Formally the types

$$
\begin{array}{ll}
\mathcal{E}^{src} ::= & \textbf{constInt} \\
& \mid \mathit{true} \\
& \mid \mathit{false} \\
& \mid \textbf{null} \\
& \mid \textbf{this} \\
& \mid \mathcal{E}^{src}\ \mathit{op}\ \mathcal{E}^{src} \\
& \mid \mathcal{E}^{src}.f \\
& \mid \textbf{var} \\
& \mid (\mathit{Class})\ \mathcal{E}^{src} \\
& \mid \mathcal{E}^{src}.m() \\
& \mid \textbf{new}\ \mathit{Class}(\mathcal{E}^{src}) \\
& \mid \mathcal{E}^{\mathcal{R}} \\
\\
\mathcal{E}^{\mathcal{R}} ::= & \mathcal{E}^{src}\ \mathcal{R}\ \mathcal{E}^{src} \\
& \mid \mathcal{E}^{src}\ \textbf{instanceof}\ \mathit{Class} \\
\\
& \mathcal{R} \in \{\leq, <, \geq, >, =, \neq\} \\
\\
\mathcal{STMT} ::= & \texttt{skip} \\
& \mid \mathcal{STMT}; \mathcal{STMT} \\
& \mid \texttt{if}\ (\mathcal{E}^{\mathcal{R}})\ \texttt{then}\ \{\mathcal{STMT}\}\ \texttt{else}\ \{\mathcal{STMT}\} \\
& \mid \texttt{if}\ (\mathcal{E}^{\mathcal{R}})\ \texttt{then}\ \{\mathcal{STMT}\} \\
& \mid \texttt{try}\ \{\mathcal{STMT}\}\ \texttt{catch}\ (\texttt{Exc}\ )\ \{\mathcal{STMT}\} \\
& \mid \texttt{try}\ \{\mathcal{STMT}\}\ \texttt{finally}\ \{\mathcal{STMT}\} \\
& \mid \texttt{throw}\ \mathcal{E}^{src} \\
& \mid \texttt{while}\ (\mathcal{E}^{\mathcal{R}})[\texttt{INV}, \texttt{modif}]\ \ \{\mathcal{STMT}\} \\
& \mid \texttt{return}\ \mathcal{E}^{src} \\
& \mid \mathcal{E}^{src} = \mathcal{E}^{src}
\end{array}
$$

Figure 7.1: SOURCE LANGUAGE

are

$$\mathit{JType} ::= \mathit{Class},\ \mathit{Class} \in\ \texttt{ClassTypes} \mid \textbf{int} \mid \texttt{boolean}$$

The source language supports also control flow constructs like the statement which does nothing `skip`, the compositional statement $\mathcal{STMT};\mathcal{STMT}$. The semantics of such a statements is that if the first statement terminates execution normally immediately after it executes the second statement. The conditional statement `if` $(\mathcal{E}^{\mathcal{R}})$ `then` $\{\mathcal{STMT}\}$ `else` $\{\mathcal{STMT}\}$ which stands for an if statement.The semantics of the construct is the standard one, i.e. if the relation expression $\mathcal{E}^{\mathcal{R}}$ evaluates to true then the statement in the `then` branch

is executed, otherwise the statement in the `else` branch is executed. Next, there is a construct for handling exceptions `try` $\{\mathcal{STMT}\}$ `catch` $(Class)$ $\{\mathcal{STMT}\}$. Its meaning is that if the statement following the `try` keyword throws an exception of type `Exc` then the exception will be caught by the statement following the `catch` keyword. The source language supports also subroutines via the statement `try` $\{\mathcal{STMT}\}$ `finally` $\{\mathcal{STMT}\}$. The meaning of the construct is that no matter how the statement following the keyword `try` terminates, the statement introduced by the keyword `finally` must execute after it. The language also provides a construct for expressing loops `while` $(\mathcal{E}^{\mathcal{R}})$[`INV`, `modif`] $\{\mathcal{STMT}\}$ states for a loop statement where the body statement $\mathcal{STMT}$ will be executed until the relational expression $\mathcal{E}^{\mathcal{R}}$ evaluates to false. Note that a loop statement is provided with a predicate `INV` which must hold whenever the loop entry is reached and with a list of expressions `modif` which give all the locations that may be modified per loop iteration. The construct `return` $\mathcal{E}^{src}$ is the statement by which method execution may terminate and control will be transfered to the method caller. The last statement that we consider is the assignment statement $\mathcal{E}^{src} = \mathcal{E}^{src}$. It states that the expression on the left of the assignment sign = is assigned the value of the expression on the right.

In Fig. 7.2, we give an example program written in our source language. It is the method square which calculates the the square of the parameter i. First, we store the absolute value of i in the variable v. Then the while statement calculates the sum of the impair positive numbers smaller than v which is the square of i. The example is also provided with specification written in JML. The specification states that the method returns the square of its parameter and that the loop invariant is (0 <= s) && (s <= v) && sqr == s*s

```
1  //@ ensures \result == i*i;
2  public int square( int i ) {
3    int sqr  = 0;
4    int v = 0;
5    if ( i < 0)
6      then {
7        v = -i;}
8      else {
9        v = i;}
10   int s = 0;
11   /*@ loop_modifies s, sqr;
12     @ loop_invariant (0 <= s) && (s <= v) && sqr == s*s ;
13     @*/
14   while( s < v ) {
15     sqr = sqr + 2*s + 1;
16     s = s+1;}
17   return sqr;}
```

Figure 7.2: METHOD SQUARE WRITTEN IN OUR SOURCE LANGUAGE

7.3 Compiler

We now turn to specify a simple compiler from the source language presented in Section 7.2 into the bytecode language.

The compiler is the triple

$$< \ulcorner *, *, * \urcorner_{\mathtt{m}}, \mathsf{addExcHandler}, \mathsf{addLoopSpec} >$$

The first component in the triple is a function $\ulcorner *, *, * \urcorner_{\mathtt{m}}$ which transforms statements and expressions into a sequence of bytecode instructions of the body of the method `m`. Note that it does not perform any optimizations.

The second component is the procedure addExcHandler which compiles the source exception handlers. Note that that the source language supports syntactic structure for encoding exception handlers while bytecode programs keep track of them via a data structure which describes how regions in the bytecode are protected from exceptions.

The third component of the compiler is the procedure addLoopSpec which manages the compilation of loop specification, namely loop invariants and the modified expressions.

The compiler presented here is realistic as the first and second component actually resemble closely a non-optimizing Java compiler. The third component is not a typical part of a Java compiler. It actually corresponds to our JML2BML compiler which compiles loop invariants into class attributes and finds their respective place in the bytecode.

In the following, in the next subsection 7.3.1 we define the procedure for the compilation of exception handlers, in subsection 7.3.2, we will present the procedure for compiling loop invariants. In subsection 7.3.3 we proceed with the definition of the compiler function $\ulcorner *, *, * \urcorner_{\mathtt{m}}$ for exressions and statements. The last subsection 7.3.4 states the properties of the bytecode produced by the compiler.

7.3.1 Exception handler table

Our source language contains exception handler constructions, and thus, when compiling a method body the compiler should keep track of the exception handlers by adding information in the exception handler table array excHndlS (presented in Section 2.3) every time it sees one. To do this, the compiler is provided with a procedure with the following name and signature:

$$\mathsf{addExcHandler} : \mathbf{Method} * nat * nat * nat * \mathbf{Class}_{exc}$$

The procedure adds a new element in the exception handler table of a method `m`. The meaning of the new element is that every exception of type `Exc` thrown in between the instructions $start \ldots end$ can be handled by the code starting at index h. The formal definition of the procedure is the following:

$$\begin{array}{l} \mathsf{addExcHandler}(\mathtt{m},\ start,\ end,\ h,\ \mathtt{Exc}) = \\ \quad \mathtt{m}.\mathsf{excHndlS} := \{(start, end, h, \mathtt{Exc}), \mathtt{m}.\mathsf{excHndlS}\} \end{array}$$

We can remark that when the procedure addExcHandler adds a new element in the exception handler table array of a method m, the new element is added at the beginning of the exception handler table m.excHndlS.

7.3.2 Compiling loop invariants

When compiling a method m, the compiler will also take care of the loop specification in the source loops by adding it in the loop specification table m.loopSpecS(defined in Section 4.4) of the method m.

This is done by the procedure addLoopSpec which has the following signature

$$\mathsf{addLoopSpec} : \mathbf{Method} * nat * P * list\ E$$

The procedure takes a method m, an index in the array of bytecode instructions i of m, a predicate INV, a list of modified expressions modif and adds a new element in the table of loop invariants m.loopSpecS of method m. Or more formally :

$$\begin{array}{l}\mathsf{addLoopSpec}(\mathtt{m},\ i,\ \mathtt{INV},\ \mathtt{modif}) = \\ \ \mathtt{m}.\mathsf{loopSpecS} := \{(i, \mathtt{INV}, \mathtt{modif}), \mathtt{m}.\mathsf{loopSpecS}\}\end{array}$$

7.3.3 Compiling source program constructs in bytecode instructions

As we stated in the beginning the function for compiling program statements and expressions in bytecode instruction is named $\ulcorner\urcorner$

$$\ulcorner\urcorner : nat * (\mathcal{STMT} \cup \mathcal{E}^{src}) * nat \longrightarrow \mathrm{I}[\ ]$$

The compiler function takes three arguments: a natural number s from which the labeling of the compilation starts, the statement $\mathcal{STMT}$ or expression $\mathcal{E}^{src}$ to be compiled and a natural number which is the greatest label in the compilation and returns an array of bytecode instructions which is the compilation of the source construct. As we shall see in the following of this section, we will define inductively the compiler over the structure of the compiled source construction.

We first look at Fig. 7.3 how expressions are compiled. From the figure, we can see that the compilation of the constants like constant integers, the boolean constants *true* , *false* , **null** are compiled with the instruction push. Note also that the boolean constants *true* and *false* are encoded as the integers 1 and 0 in the bytecode. The field access expression $\mathcal{E}^{src}.f$ compilation consists in the compilation $\ulcorner s, \mathcal{E}^{src}, e-1\urcorner_{\mathtt{m}}$ of the expression $\mathcal{E}^{src}$ followed by the instruction e : getfield f which is the last instruction of the compilation. May be more attention deserve the compilation of the instance creation expression **new** Cl($\mathcal{E}^{src}$). The first instruction in the compilation is the instruction s : new Cl which creates a new reference in the heap and pushes it on the stack top (see for the operational semantics of the instruction in Section 2.8). The next instruction in the compilation is the instruction dupwhich duplicates the

stack top element. Then follows the compilation $\ulcorner s+2, \mathcal{E}^{src}, e-1 \urcorner_{\mathtt{m}}$; of the argument passed to the class constructor. The last instruction of the compilation is the invokation of the class constructor $\mathsf{constr}(\mathtt{Cl})$. Note that this compilation follows closely the JVM specification (see [49], Section 7.8) which mandates how standard Java compilers compiler instance creation expressions.

$$
\begin{array}{lcl}
\ulcorner s, \mathbf{constInt}, s \urcorner_{\mathtt{m}} & = & s : \text{push } \mathbf{constInt} \\
\ulcorner s, true, s \urcorner_{\mathtt{m}} & = & s : \text{push } 1 \\
\ulcorner s, false, s \urcorner_{\mathtt{m}} & = & s : \text{push } 0 \\
\ulcorner s, \mathbf{null}, s \urcorner_{\mathtt{m}} & = & s : \text{push } \mathbf{null} \\
\ulcorner s, \mathbf{this}, s \urcorner_{\mathtt{m}} & = & s : \text{load } \mathbf{reg}(0) \\
\ulcorner s, \mathcal{E}^{src}.f, e \urcorner_{\mathtt{m}} & = & \ulcorner s, \mathcal{E}^{src}, e-1 \urcorner_{\mathtt{m}}; \\
 & & e : \text{getfield } f \\
\ulcorner s, \mathcal{E}_1^{src}.m(\mathcal{E}_2^{src}), e \urcorner_{\mathtt{m}} & = & \ulcorner s, \mathcal{E}_1^{src}, e' \urcorner_{\mathtt{m}}; \\
 & & \ulcorner e'+1, \mathcal{E}_2^{src}, e-1 \urcorner_{\mathtt{m}}; \\
 & & e : \text{invoke } m \\
\ulcorner s, \mathbf{var}, s \urcorner_{\mathtt{m}} & = & s : \text{load } \mathbf{reg}(i) \\
\ulcorner s, \mathcal{E}_1^{src} \ op \ \mathcal{E}_2^{src}, e \urcorner_{\mathtt{m}} & = & \ulcorner s, \mathcal{E}_1^{src}, e' \urcorner_{\mathtt{m}}; \\
 & & \ulcorner e'+1, \mathcal{E}_2^{src}, e-1 \urcorner_{\mathtt{m}}; \\
 & & e : \text{arith\_op} \\
\ulcorner s, (\mathtt{Cl})\ \mathcal{E}^{src}, e \urcorner_{\mathtt{m}} & = & \ulcorner s, \mathcal{E}^{src}, e-1 \urcorner_{\mathtt{m}}; \\
 & & e : \text{checkcast } \mathtt{Cl}; \\
\ulcorner s, \mathcal{E}^{src} \ \mathbf{instanceof}\ \mathtt{Cl}, e \urcorner_{\mathtt{m}} & = & \ulcorner s, \mathcal{E}^{src}, e-1 \urcorner_{\mathtt{m}}; \\
 & & e : \text{instanceof } \mathtt{Cl}; \\
\ulcorner s, \mathbf{new}\ \mathtt{Cl}(\mathcal{E}^{src}), e \urcorner_{\mathtt{m}} & = & s : \text{new } \mathtt{Cl}; \\
 & & s+1 : \text{dup}; \\
 & & \ulcorner s+2, \mathcal{E}^{src}, e-1 \urcorner_{\mathtt{m}}; \\
 & & e : \text{invoke } \mathsf{constr}(\mathtt{Cl});
\end{array}
$$

Figure 7.3: Definition of the compiler for expressions

Fig. 7.4 presents the definition of the compiler function for statements which does not affect the exception handler table neither have affect the specification tables. Let us explain in more detail the rule for conditional statement. First the conditional expression is compiled $\ulcorner s, \mathcal{E}^{\mathcal{R}}, e' \urcorner_{\mathtt{m}};$. Afterwards, comes the conditional branch instruction $e'+1$: if_cond $e''+2;$. Remind that the if_cond

instruction will compare the two stack top elements (w.r.t. some condition) and if they fulfill the condition in question, the control will be transfered to instruction at index $e'' + 2$, otherwise the next instruction at index $e' + 2$ will be executed. Note that at index $e'' + 2$ starts the compilation of the then branch $\ulcorner e'' + 2, \mathcal{STMT}_1, e \urcorner_{\mathtt{m}}$; and at index $e' + 2$ starts the compilation of the else branch $\ulcorner e' + 2, \mathcal{STMT}_2), e'' \urcorner_{\mathtt{m}}$. After the compilation of the else branch follows a $e'' + 1$: goto$e + 1$ instruction which jumps at the first instruction outside the compilation of the branch statement. It deserves may be our attention that we have two different compilation rules for the assignment. More particularly, it depends on if it is a field assignment or local variable assignment. In the case for field assignment $\mathcal{E}_1^{src}.f = \mathcal{E}_2^{src}$, the last instruction of the compilation is a putfield f instruction. The compilation of a local variable assignment $\mathbf{var} = \mathcal{E}^{src}$ statement terminates with a store insruction.

est-ce que il faut changer le langage source pour avoir aussi 2 types d'affectation

$\ulcorner s, \mathcal{STMT}_1; \mathcal{STMT}_2, e \urcorner_{\mathtt{m}} =$
$\quad \ulcorner s, \mathcal{STMT}_1, e' \urcorner_{\mathtt{m}};$
$\quad \ulcorner e' + 1, \mathcal{STMT}_2, e \urcorner_{\mathtt{m}}$

$\ulcorner s, \mathtt{if}\ (\mathcal{E}^{\mathcal{R}})\ \mathtt{then}\ \{\mathcal{STMT}_1\}\ \mathtt{else}\ \{\mathcal{STMT}_2\}, e \urcorner_{\mathtt{m}} =$
$\quad \ulcorner s, \mathcal{E}^{\mathcal{R}}, e' \urcorner_{\mathtt{m}};$
$\quad e' + 1 :$ if_cond $e'' + 2;$
$\quad \ulcorner e' + 2, \mathcal{STMT}_2), e'' \urcorner_{\mathtt{m}}$
$\quad e'' + 1 :$ goto $e + 1;$
$\quad \ulcorner e'' + 2, \mathcal{STMT}_1, e \urcorner_{\mathtt{m}};$

$\ulcorner s, \mathcal{E}_1^{src}.f = \mathcal{E}_2^{src}, e \urcorner_{\mathtt{m}} =$
$\quad \ulcorner s, \mathcal{E}_1^{src}, e' \urcorner_{\mathtt{m}};$
$\quad \ulcorner e' + 1, \mathcal{E}_2^{src}, e - 1 \urcorner_{\mathtt{m}};$
$\quad e :$ putfield $f;$

$\ulcorner s, \mathbf{var} = \mathcal{E}^{src}, e \urcorner_{\mathtt{m}} =$
$\quad \ulcorner s, \mathcal{E}^{src}, e - 1 \urcorner_{\mathtt{m}}$
$\quad e :$ store $\mathbf{reg}(i);$

$\ulcorner s, \text{athrow}\ \mathcal{E}^{src}, e \urcorner_{\mathtt{m}} =$
$\quad \ulcorner s, \mathcal{E}^{src}, e - 1 \urcorner_{\mathtt{m}};$
$\quad e :$ athrow;

$\ulcorner s, \mathtt{return}\ \mathcal{E}^{src}, e \urcorner_{\mathtt{m}} =$
$\quad \ulcorner s, \mathcal{E}^{src}, e - 1 \urcorner_{\mathtt{m}};$
$\quad e :$ return

Figure 7.4: Definition of the compiler for statements

Fig. 7.5 shows the compiler definition for statements whose compilation will change the the exception handler of the current method. The first such statement is the try catch statement. The compiler compiles the normal statement $\mathcal{STMT}_1$ and the exception handler $\mathcal{STMT}_2$. Note that in the exception handler table of the bytecode representation of method $\mathtt{m}$, a new line is added

which states that the if one of bytecode instructions from index s to index e' throw a not handled exception of type *ExcType*control will be transfered to the instruction at index *ExcType*.

As you can notice, Fig 7.5 contains the compiler definition for try finally statements. We compile the finally statement $\mathcal{STMT}_2$ by inlining, it is first compiled as a code executed after $\mathcal{STMT}_1$ and then it is compiled as part of the default exception handler. The default exception handler which starts at index $e'' + 2$ and which will handle any exception thrown by $\ulcorner s, \mathcal{STMT}_1, e' \urcorner_{\mathtt{m}}$. The exception handler first stores the thrown exception in the local variable at index l, then executed the subroutine code and after the execution rethrows the exception stored in the local variable l.

This compilation differs from the compilation scheme in the JVM specification for finally statements, which requires that the subroutines must be compiled using jsr and ret instructions. However, the semantics of the programs produced by the compiler presented here and a compiler which follows closely the JVM specification is equivalent. In the following, we discuss informally why this is true. The semantics of a jsr k insruction is to jump to the first instruction of the compiled subroutine which starts at index k and pushes on the operand stack the index of the next instruction of the jsr that caused the execution of the subroutine. The first instruction of the compilation of the subroutine stores the stack top element in the local variable at index k (i.e. stores in the local variable at index k the index of the instruction following the jsr instruction). Thus, after the code of the subroutine is executed, the ret k instruction jumps to the instruction following the corresponding jsr. This behaviour can be actually simulated by programs without jsr and ret but which inline the subroutine code at the places where a jsr to the subroutine is done.

Note:

1. we assume that the local variable l is not used in the compilation of the statement $\mathcal{STMT}_2$, which guarantees that after any execution which terminates normally of $\ulcorner e'' + 3, \mathcal{STMT}_2, e - 2 \urcorner_{\mathtt{m}}$ the local variable l will still hold the thrown object

2. here we also assume that the statement $\mathcal{STMT}_1$ does not contain a return instruction

The last remark that we would like to make is that subroutines and their compilation via ret and jsrhas always presented a problem for Java. First, they slow down the performance of the JVM because of the special way ret and jsr work. Second, the analysis performed by the bytecode verifier in the JVM becomes rather complex because (and its first version was erroneus) of the Java subroutines. In the last version of Java Sun compiler, subroutines has become obsolete where they are compiled by inlining. Thus, the compiler presented here represents a realistic approximation of the Java Sun compiler.

verify this

We have put separately in Fig.7.6 the compiler definition for loop statements as its compilation is particular because it is the unique case where the specification tables of the current method are affected. Particularly, the compiler adds a

$$
\begin{array}{l}
\ulcorner s, \texttt{try}\ \{\mathcal{STMT}_1\}\ \texttt{catch}\ (\mathit{ExcType}\ \textbf{var})\{\mathcal{STMT}_2\}, e\urcorner_{\texttt{m}} = \\
\quad \ulcorner s, \mathcal{STMT}_1, e'\urcorner_{\texttt{m}}; \\
\quad e'+1 : \text{goto } e+1; \\
\quad \ulcorner e'+2, \mathcal{STMT}_2, e\urcorner_{\texttt{m}}; \\
\\
\quad \mathsf{addExcHandler}(\texttt{m},\ s,\ e',\ e'+2,\ \mathit{ExcType})) \\
\\
\ulcorner s, \texttt{try}\ \{\mathcal{STMT}_1\}\ \texttt{finally}\ \{\mathcal{STMT}_2\}, e\urcorner_{\texttt{m}} = \\
\quad \ulcorner s, \mathcal{STMT}_1, e'\urcorner_{\texttt{m}}; \\
\quad \ulcorner e'+1, \mathcal{STMT}_2, e''\urcorner_{\texttt{m}}; \\
\quad e''+1 : \text{goto } e+1; \\
\\
\quad \{\text{ default exception handler}\} \\
\quad e''+2 : \text{store } l; \\
\quad \ulcorner e''+3, \mathcal{STMT}_2, e-2\urcorner_{\texttt{m}}; \\
\quad e-1 : \text{load } l; \\
\quad e : \text{athrow}; \\
\\
\quad \mathsf{addExcHandler}(\texttt{m},\ s,\ e',\ e''+2,\ \texttt{Exception})
\end{array}
$$

Figure 7.5: Definition of the compiler for statements that change the exception handler table

new element in the table of loop invariants of the method `m`. This new element relates the index $e'+1$ with the loop invariant `INV` and the list of modified expressions `modif`.

$$
\begin{array}{l}
\ulcorner s, \texttt{while}\ (\mathcal{E}^{\mathcal{R}})[\texttt{INV}, \texttt{modif}]\ \ \{\mathcal{STMT}\}, e\urcorner_{\texttt{m}} = \\
\quad s : \text{goto } e'+1; \\
\quad \ulcorner s+1, \mathcal{STMT}, e'\urcorner_{\texttt{m}}; \\
\quad \ulcorner e'+1, \mathcal{E}^{\mathcal{R}}, e-1\urcorner_{\texttt{m}}; \\
\quad e : \text{if\_cond } s+1; \\
\\
\quad \mathsf{addLoopSpec}(\texttt{m},\ e'+1,\ \texttt{INV},\ \texttt{modif})
\end{array}
$$

Figure 7.6: Definition of the compiler for the loop statement

For an illustration, we can turn to Fig. 7.7. The example shows the correspondance the bytecode (left) resulting from the compilation described above (on the left) and the source lines(right) of method square from Fig. 7.2.

We can see in the figure how the branch statement is compiled (bytecode instructiosn from 4 to 11). It also shows us the somewhat unusual way into which the while statement is compiled. More particularly, the compilation of the test of the **while** is after its body while semantically the test must be executed at every iteration before the loop body. The compilation is actually correct because the instruction 14 **goto** 27 jumps to the compilation of the line **while**(s<v) and thus, the execution proceeds in the expected way. We have

also marked the instructions which correspond to the loop entry and loop end which are the instructions at index 27 and 26.

Note that the bytecode has been generated by a standard Java compiler. We have only modified the compilation of s = s+1; to match the definition of our compiler[1]. Note also that we keep the same names on bytecode and source. This is done for the sake of simplicity and in this chapter, we shall use always this convention.

7.3.4 Properties of the compiler function

In this subsection, we will focus on the properties of the bytecode produced by the compiler presented above. These properties although a straightforward consequence of the compiler definition are actually important for establishing formally the equivalence between source and bytecode proof obligations.

In the following, we use the notation $\mathcal{S}$ when we refer both to statements $\mathcal{STMT}$ and $\mathcal{E}^{src}$.

The first property that we observe is that the last instruction e in the compilation $\ulcorner s, \mathcal{STMT}, e \urcorner_{\mathtt{m}}$ of a statement $\mathcal{STMT}$ is always in execution relation (see Fig. 2.9 for the definition of execution relation between instructions) with the instruction $e+1$.

In order to get a precise idea of what we mean, the reader may take a look at the example in Fig. 7.7. There, we can see that the last instruction of the compilation of the statement **int** sqr = 0; is the instruction 1 **store** sqr and that it is in execution relation with the instruction at index 2 **const** 0. The same holds also for the compilation of the **if** statement where the last instruction in its compilation is 11 **store** v and is in execution relation with 12 **const** 0. Actually, the compilation of every statement or expression in the example has this property.

Property 7.3.4.1 (Compilation of statements and expressions) *For any statement or expression $\mathcal{S}$ which does not terminate on* return*, start label s and end label e, the compiler will produce a list of bytecode instruction $\ulcorner s, \mathcal{STMT}, e \urcorner_{\mathtt{m}}$ such that*

$$e \rightarrow e+1$$

The proof is trivial and is by case analysis of the compiled statements.

Let us now turn to the next property. Informally, it states that if there are instructions inside a compiled statetement or expression $\ulcorner s, \mathcal{S}, e \urcorner_{\mathtt{m}}$ which are in execution relation[2] with an instruction k which is not the start of an exception handler and which is outside the compilation $\ulcorner s, \mathcal{S}, e \urcorner_{\mathtt{m}}$ of $\mathcal{S}$ then $k = e+1$. The conditions $\neg(k \in \ulcorner s, \mathcal{S}, e \urcorner_{\mathtt{m}})$ and $\neg \mathit{isExcHandlerStart}(k)$ eliminate the case when the execution relation is between an instruction inside the compilation $\ulcorner s, \mathcal{S}, e \urcorner_{\mathtt{m}}$ which may throw an exception and the start instruction of the proper handler exception handler. For illustration, we may consider Fig. 7.7 and see that the

[1] the Java compiler will tend to compile incrementation into **iinc**

[2] see definition in Fig. 2.9

```
                    int square( int i )
 0 const 0          int sqr  = 0;
 1 store sqr

 2 const 0          int v = 0;
 3 store v

 4 load i           if ( i >= 0)
 5 ifge 10

 6 load i               else{v =−i;}
 7 neg
 8 store v

 9 goto 12

10 load i               then{v = i;}
11 store v

12 const 0          int s = 0;
13 store s

14 goto 27

15 load sqr            sqr = sqr + 2∗s + 1;
16 const 2
17 load s
18 mul
19 add
20 const 1
21 add
22 store sqr

23 load s              s = s+1;
24 const 1
25 add
26 store s          LOOP END

27 load s           LOOP START while( s < v )
28 load v
29 if_icmplt 15

30 load sqr         return sqr;
31 return
```

Figure 7.7: RELATION BETWEEN BYTECODE AND SOURCE CODE OF METHOD SQUARE FROM FIG. 7.2

compilation of the statement **if** (i >= 0) **then** {v = i} **else** {v = −i} which comprises instructions from 4 to 11. Thus, the compilation contains two jump instructions. The first is the instruction 5 **ifge** 10 which jumps inside the compilation of the if statement and the instruction 9 **goto** 12 which jumps apparently

to instruction 12. We can now state this property formally.

Property 7.3.4.2 (Compilation of statements and expressions) *For any statement or expression $\mathcal{S}$, start label s and end label e, the compiler will produce a list of bytecode instruction $\ulcorner s, \mathcal{S}, e \urcorner_{\mathtt{m}}$ such that:*

$$\begin{array}{l} \forall i, (i \in \ulcorner s, \mathcal{S}, e \urcorner_{\mathtt{m}}) \wedge \\ \quad (i \rightarrow k) \wedge \\ \quad \neg(k \in \ulcorner s, \mathcal{S}, e \urcorner_{\mathtt{m}}) \\ \quad \neg \mathit{isExcHandlerStart}(k) \Rightarrow \\ \qquad k = e + 1 \end{array}$$

The next three properties deal with the substatement relation. In the following, for denoting that $\mathcal{S}'$ is a substatement of $\mathcal{S}$ (i.e.$\mathcal{S}'$ is contained in $\mathcal{S}$) we shall use the notation $\mathcal{S}[\mathcal{S}']$. For denoting that $\mathcal{S}'$ is a strict substatement of $\mathcal{S}$ (i.e. that $\mathcal{S}'$ is contained in $\mathcal{S}$ and there is no $\mathcal{S}''$ such that $\mathcal{S}''$ is contained in $\mathcal{S}$ and $\mathcal{S}'$ is contained in $\mathcal{S}''$) we use the notation $\mathcal{S}[[\mathcal{S}']]$.

The next property of the compiler is that any statement or expression is compiled in a list of bytecode instructions such that there could not be jumps from outside inside the compilation of a statement or expression, i.e. the control flow can reach the instructions representing the compilation $\ulcorner s, \mathcal{S}, e \urcorner_{\mathtt{m}}$ of statement $\mathcal{S}$ only by passing through the beginning of the compilation, i.e. the instruction at index s. For instance, the instructions in the compilation of any statement(expression) in Fig. 7.7 can be reached from outside of the statement(expression) compilation only via the first instruction of the statement(expression) compilation.

Property 7.3.4.3 (Compilation of statements and expressions) *For all statements $\mathcal{S}'$ and $\mathcal{S}$, such that $\mathcal{S}[\mathcal{S}']$ and their compilations are $\ulcorner s, \mathcal{S}, e \urcorner_{\mathtt{m}}$ and $\ulcorner s', \mathcal{S}', e' \urcorner_{\mathtt{m}}$ then :*

$$\begin{array}{l} (i_j \in \ulcorner s, \mathcal{S}[\mathcal{S}'], e \urcorner_{\mathtt{m}} \wedge \\ \neg(i_j \in \ulcorner s', \mathcal{S}', e' \urcorner_{\mathtt{m}}) \wedge \\ i_k \in \ulcorner s', \mathcal{S}', e' \urcorner_{\mathtt{m}} \wedge \\ i_j \rightarrow i_k) \Rightarrow \\ \qquad s' = k \end{array}$$

The proof is by induction over the structure of statements and expressions and uses the previous lemma 7.3.4.2.

The next lemma states that the substatement relation on source expressions and statements is preserved by the compiler. For instance, the compilation of the loop body (comprising instructions from 15 to 26) in Fig. 7.7 is part of the compilation of the loop itself (comprising instructions from 14 to 29) .

Property 7.3.4.4 (Substatement and subexpression relation preserved) *For all statements $\mathcal{S}'$ and $\mathcal{S}$ with respective compilations are $\ulcorner s', \mathcal{S}', e' \urcorner_{\mathtt{m}}$ and $\ulcorner s, \mathcal{S}, e \urcorner_{\mathtt{m}}$ if $\mathcal{S}[\mathcal{S}'] \Leftrightarrow \ulcorner s', \mathcal{S}', e' \urcorner_{\mathtt{m}} \in \ulcorner s, \mathcal{S}, e \urcorner_{\mathtt{m}}$*

This follows directly from the compiler function definition.

The next property states that if the compilations of two statements in a method body share instructions then either the compilation of one of them is completely inside of the other or viceversa. This is also evident from Fig. 7.7.

Property 7.3.4.5 (No overlapping compilation) *For all statements $\mathcal{S}_1$ and $\mathcal{S}_2$ such that their compilations are $\ulcorner s_1, \mathcal{S}_1, e_1 \urcorner_{\mathtt{m}}$ and that $\ulcorner s_2, \mathcal{S}_2, e_2 \urcorner_{\mathtt{m}}$, if we have that $\exists k, s_1 \leq k \leq e_1 \wedge s_2 \leq k \leq e_2$ then the following holds:*

$$\begin{array}{l} \ulcorner s_2, \mathcal{S}_2, e_2 \urcorner_{\mathtt{m}} \in \ulcorner s_1, \mathcal{S}_1, e_1 \urcorner_{\mathtt{m}} \\ \vee \\ \ulcorner s_1, \mathcal{S}_1, e_1 \urcorner_{\mathtt{m}} \in \ulcorner s_2, \mathcal{S}_2, e_2 \urcorner_{\mathtt{m}} \end{array}$$

Next, we give a definition for a set of instructions such that they execute sequentially which will be used for establishing afterwards the properties of the bytecode instructions resulting in expression compilation.

Definition 7.3.4.1 (Block of instructions) *If the list of instructions $l = [i_1 \dots i_n]$ in the compilation of method $\mathtt{m}$is such that*

- *none of the instructions is a target of an instruction i_j which does not belong to l except for i_1*
- *none of the instructions in the set is a jump instruction, i.e. $\forall m, m = 1..k \Rightarrow \neg(i_m \in \{\text{goto}, \text{if\_cond}\})$*
- *$\forall j, i_1 < j \leq i_n, \neg\exists k \in \mathtt{m.body}, k \rightarrow^l j$*

We denote such a list of instructions with $i_1; ...; i_k$

The next lemma states that the compilation of an expression $\mathcal{E}^{src}$ results in a block of bytecode instructions. For instance, consider the compilation of the expression sqr + 2∗s + 1; in Fig. 7.7 comprised between instructions 16-21. Instructions 16-21 satisfy the three points from the above definition.

Property 7.3.4.6 (Compilation of expressions) *For any expression $\mathcal{E}^{src}$, starting label s and end label e, the compilation $\ulcorner s, \mathcal{E}^{src}, e \urcorner_{\mathtt{m}}$ is a block of bytecode instruction in the sense of Def. 7.3.4.1*

The following two statements concern the loops on bytecode and source. In particular, we want that a cycle appears in the compilation of a statement $\mathcal{STMT}$ only if it contains a loop (or is itself a loop). For instance, we can see in Fig. 7.7 that the unique cycle in the bytecode corresponds to the source loop and that the instruction marked with LOOP END and the instruction marked with LOOP START correspond respectively to the end instruction in the compilation of the source loop body and to start instruction in the compilation of the compilation of the **while** test. Stated formally, we get the following property.

alternative formulation: that the only loop entri the first instructions of a body

Property 7.3.4.7 (Cycles in the control flow graph) *The compilation* $\ulcorner s, \mathcal{STMT}, e \urcorner_{\mathtt{m}}$ *of a* $\mathcal{STMT}$ *may contain an instruction* k *and* j *which are respectively a loop entry and a loop end in the sense of Def. 2.9.1 (i.e. there exists* j *such that* $j \rightarrow^l k$ *) only if* $\mathcal{STMT}$ *is a loop or contains a substatement* $\mathcal{STMT}'$ *which is a loop statement and the following holds:*

$$\mathcal{STMT} =$$
$$\mathtt{while}\ (\mathcal{E}^{\mathcal{R}})[\mathtt{INV}, \mathtt{modif}]\ \ \{\mathcal{STMT}\}$$

$$\vee$$

$$\mathcal{STMT}[\mathcal{STMT}'] \wedge$$
$$\mathcal{STMT}' =$$
$$\mathtt{while}\ (\mathcal{E}^{\mathcal{R}})[\mathtt{INV}, \mathtt{modif}]\ \ \{\mathcal{STMT}\}$$

$$\ulcorner s, \mathtt{while}\ (\mathcal{E}^{\mathcal{R}})[\mathtt{INV}, \mathtt{modif}]\ \ \{\mathcal{STMT}\}, e \urcorner_{\mathtt{m}} =$$
$$s : \text{goto } e' + 1;$$
$$\ulcorner s + 1, \mathcal{STMT}, e' \urcorner_{\mathtt{m}};$$
$$\ulcorner e' + 1, \mathcal{E}^{\mathcal{R}}, e - 1 \urcorner_{\mathtt{m}};$$
$$e : \text{if\_cond } s + 1;$$

$$\Rightarrow k = e' + 1 \wedge j = e'$$

Another property concerning cycles in the control flow graph is that all the instructions in a compilation$\ulcorner s, \mathcal{STMT}, e \urcorner_{\mathtt{m}}$ of a statement $\mathcal{STMT}$which target the instruction $e + 1$ are in the same execution relation with $e + 1$,i.e. if $e + 1$ is a loop entry either all are loop ends or none of them is:

Property 7.3.4.8 (Cycles in the control flow graph) *For every statement* $\mathcal{STMT}$ *its compilation* $\ulcorner s, \mathcal{STMT}, e \urcorner_{\mathtt{m}}$ *is such that* $(\exists k, s \leq k \leq e, k \rightarrow^l e + 1) \iff (\forall k, s \leq k \leq e, k \rightarrow e + 1 \Rightarrow k \rightarrow^l e + 1)$

Next, we shall focus on properties which concern the compilation of exception handlers. As we saw in the previous section, the compiler keeps track of the exception handlers by adding them in the exception handler table.

We illustrate this by the example in Fig. 7.8 which shows both the bytecode (on the left) and source code(on the right) of . the method abs. The method abs gets as parameter n an object of type Number. The method returns the absolute value of the integer field n.value of the parameter n. The absolute value is stored in the method local variable abs. This is implemented by the if statement. However, in the test of the if statement, we dereference the parameter n ignoring whether it is **null** or not. That is why the **if** statement may throw a `NullPntrExc` and thus, it is wrapped in a **try catch** block. The exception handler (the statement following the **catch** keyword) creates a new instance of class Number and initializes its value with 0. Finally the method returns the absolute value of the parameter.

The exception handler table abs.**ExcHandler** contains one element which describes the unique exception handler in the method. In particular, it states that the region between 2 and 20 is protected from `NullPntrExc` by the bytecode starting at index 22. We may remark that the region between 2 and 20 corresponds to the compilation of the **if** statement.

```
int abs(Number reg(1))                int abs(Number n )
0 const 0                             int abs = 0;
1 store abs

                                      try {
2 load n                                 if (n.value >= 0)
3 getfield Number.value
4 iflt 9

5 load n                                    else {abs = − n.value;}
6 getfield Number.value
7 store abs

8 goto 21

9 load n                                    then {abs = n.value;}
18 getfield Number.value
19 neg
20 store abs

21 goto 29
                                      }
                                      catch(NullPointerExc e) {
22 store e                               n = new Number(0);
23 new <Number>
24 dup
25 const 0
26 invoke Number.init
27 store n                            }

28 load abs                           return abs
29 return

abs.ExcHandler=
  startPc     =   2
  endPc       =   20
  handlerPc   =   22
  exc         =    NullPntrExc
```

Figure 7.8: RELATION BETWEEN BYTECODE AND SOURCE CODE OF METHOD ABS

This illustrates the next property of the compiler. More particularly, it

states that the fields startPc and endPc in an element of the exception handler table are the start and end index of the compilation of a source statement in the original source program.

Property 7.3.4.9 (Exception handler element corresponds to a statement) *Every element* $(s, e, eH, \texttt{Exc})$ *in the exception handler table* m.excHndlS *is such that exists a statement* $\mathcal{STMT}$ *such that* $\ulcorner s, \mathcal{STMT}, e \urcorner_{\texttt{m}}$

Proof: This follows directly from the definition of the compiler. The proof is done by contradiction. From the compiler definition, we get that elements are added in m.excHndlS only in the cases of try catch and try finally statement compilation and that the guarded regions in the added elements correspond to statements.
Qed.

The following property states that the exceptions thrown by a statement which is not a try catch and the exceptions thrown by its strict substatement will be handled by the same exception handler, i.e. the result of the function *findExcHandler*(*,*,*) will be the same if we pass as an argument their respective last instructions.

For instance, consider in Fig. 7.8 the bytecode version of the program. The last instruction in the compilation of the **if** statement is the instruction at index 20 and the last instruction in the compilation of the **else** branch is 7. For any exception type Exc, the application *findExcHandler*(Exc ,20,abs.**ExcHandler**) returns the same value as *findExcHandler*(Exc ,7,abs.**ExcHandler**).

Property 7.3.4.10 (Exception handler property for statements) *For every statement* $\mathcal{STMT}$ *which is not a try catch statement in method* m *and its strict substatement* $\mathcal{STMT}'$*, i.e.* $\mathcal{STMT}[[\mathcal{STMT}']]$ *if their respective compilations are* $\ulcorner s, \mathcal{STMT}, e \urcorner_{\texttt{m}}$ *and* $\ulcorner s', \mathcal{STMT}', e' \urcorner_{\texttt{m}}$ *then the following holds:*

$$\forall \texttt{Exc}, \mathit{findExcHandler}(\texttt{Exc}, e, \texttt{m.excHndlS}) = \mathit{findExcHandler}(\texttt{Exc}, e', \texttt{m.excHndlS})$$

A similar property can be established for expressions.

Property 7.3.4.11 (Exception handler property for expressions) *For every statement* $\mathcal{STMT}$ *which is not a try catch in method* m *and its strict subexpression* $\mathcal{E}^{src}$*, i.e.* $\mathcal{STMT}[[\mathcal{E}^{src}]]$ *we have that if their respective compilations are* $\ulcorner s, \mathcal{STMT}, e \urcorner_{\texttt{m}}$ *and* $\ulcorner s', \mathcal{E}^{src}, e' \urcorner_{\texttt{m}}$ *and (2) then the following holds:*

$$\forall \texttt{Exc}, \forall\ i, s' \leq i \leq e', \mathit{findExcHandler}(\texttt{Exc}, e, \texttt{m.excHndlS}) = \mathit{findExcHandler}(\texttt{Exc}, i, \texttt{m.excHndlS})$$

The next property states that a try catch statement is such that try $\{\mathcal{STMT}_1\}$ catch (*ExcType* **var**)$\{\mathcal{STMT}_2\}$ any exception except *ExcType* thrown by the last instruction in the compilation of $\mathcal{STMT}_1$ or the last instruction of the compilation of $\mathcal{STMT}_2$ are handled by the same exception handler. In particular, if the last instruction in the compilation of $\mathcal{STMT}_1$ throws an exception

of type *ExcType* it will be handled by the start instruction of the compilation of $\mathcal{STMT}_2$.

For instance, the last instruction in the compilation of the the try catch statement in Fig. 7.8 is the instruction at index 27 and the last instruction in the compilation of the try statement is the instruction at index 20. We can see that any excepition except `NullPntrExc` which might be thrown from these instructions will be handled in the same way. This in particular is evident if we look at the exception handler table.

Property 7.3.4.12 (Exception handlers and try catch statements) *For every try catch statement* `try` $\{\mathcal{STMT}_1\}$ `catch` (*ExcType* **var**)$\{\mathcal{STMT}_2\}$ *its compilation*

$$\begin{array}{l}\ulcorner s, \mathcal{STMT}_1, e' \urcorner_{\mathtt{m}};\\ e'+1 : \text{goto } e+1;\\ \ulcorner e'+2, \mathcal{STMT}_2, e \urcorner_{\mathtt{m}};\end{array}$$

is such that the following holds

$$\begin{array}{l}\forall \mathtt{Exc}, \neg(\mathtt{Exc} <: \mathit{ExcType}) \Rightarrow \begin{array}{l}\mathit{findExcHandler}(\mathtt{Exc}, e', \mathtt{m}.\mathsf{excHndlS}) =\\ \mathit{findExcHandler}(\mathtt{Exc}, e, \mathtt{m}.\mathsf{excHndlS})\end{array}\\ \wedge\\ \mathit{findExcHandler}(\mathit{ExcType}, e', \mathtt{m}.\mathsf{excHndlS}) = e'+2\end{array}$$

7.4 Weakest precondition calculus for source programs

7.4.1 Source assertion language

The properties that our predicate calculus treats are from first order predicate logic. In Fig. 7.9, we give the formal definition of the assertion language into which the properties are encoded. Note that the language is almost the same as the bytecode assertion language presented earlier (see Chapter 4). However, the assertion language for source programs does not support stack expressions ($\mathbf{st}(\mathbf{cntr} + - \ldots)$ and **cntr**) which make sense only for bytecode.

7.4.2 Weakest predicate transformer for the source language

The weakest precondition calculates the predicate weakest precondition predicate WP statement $\mathcal{STMT}$ in method `m` from our source language, for any normal postcondition nPost^{src} and exceptional postcondition function $\mathsf{excPost}^{src}$, such that if it holds in the pre state of $\mathcal{STMT}$ and if $\mathcal{STMT}$ terminates normally then nPost^{src} holds in the poststate and if $\mathcal{STMT}$ terminates on exception Exc then $\mathsf{excPost}^{src}(\mathit{Exc}, \mathcal{STMT})$ holds. In the following, in subsection 7.4.2 we discuss how the exceptional postcondition function is defined.

$$
\begin{array}{lll}
\mathcal{F} ::= & & \mathcal{E}^{spec}\mathcal{R}\mathcal{E}^{spec} \\
& | & \textbf{instances}(\mathcal{E}^{spec}) \\
& | & \textbf{true} \\
& | & \textbf{false} \\
& | & \mathcal{F} \wedge \mathcal{F} \\
& | & \mathcal{F} \vee \mathcal{F} \\
& | & \mathcal{F} \Rightarrow \mathcal{F} \\
& | & \forall x(\mathcal{F}(x)) \\
& | & \exists x(\mathcal{F}(x)) \\
\\
\mathcal{R} ::= & & == | \neq | \leq | \leq | \geq | > | <: \\
\\
\mathcal{E}^{spec} ::= & & \textbf{constInt} \\
& | & \textit{true} \\
& | & \textit{false} \\
& | & \textbf{bv} \\
& | & \backslash\textbf{EXC} \\
& | & \mathcal{E}^{spec}\ op\ \mathcal{E}^{spec} \\
& | & \mathcal{E}^{spec}.f \\
& | & \textbf{var} \\
& | & \textbf{null} \\
& | & \textbf{this} \\
& | & \backslash\textbf{typeof}(\mathcal{E}^{spec}) \\
& | & \backslash\textbf{result}
\end{array}
$$

Figure 7.9: SOURCE ASSERTION GRAMMAR

Subsections 7.4.2 and 7.4.2 present respectively the definition of *wp* function for expressions and statements.

Exceptional Postcondition Function

As we stated earlier the weakest predicate transformer manages both the normal and exceptional termination of an expression(statement). In both cases the expression(statement) has to satisfy some condition : the normal postcondition in case of normal termination and the exceptional postcondition for exception `Exc` if it terminates on exception `Exc`

We introduce a function $\mathsf{excPost}^{src}$ which maps exception types to predicates

$$\mathsf{excPost}^{src} : ExcType \longrightarrow P$$

The function $\mathsf{excPost}^{src}$ returns the predicate $\mathsf{excPost}^{src}(\texttt{Exc})$ that must

hold in a particular program point if at this point an exception of type `Exc` is thrown.

We also use function updates for $\mathsf{excPost}^{src}$ which are defined in the usual way

$$\mathsf{excPost}^{src}[\oplus \texttt{Exc'} \ \rightarrow P](\texttt{Exc}) = \begin{cases} P & if \texttt{Exc} <: \texttt{Exc'} \\ \mathsf{excPost}^{src}(\texttt{Exc}) & else \end{cases}$$

Expressions

In the following, we shall give the definition of a weakest predicate transformer function for expressions.

As we shall see, for some of the expressions the definition of the weakest precondition function is trivial (i.e. it is the identity function) as they do have no side effects. However, this is not the case for expressions that might throw an exception or which may change values of program variables. The predicate returned by the weakest precondition predicate transformer for expressions which may throw an exception will basically acumulate the hypothesis under which the evaluation of the expression terminates normally and the conditions under which it terminates exceptionally.

The weakest precondition function for expressions has the following signature:

$$wp^{src} : \mathcal{E}^{src} \rightarrow \mathcal{F} \rightarrow (\ \texttt{Exc} \rightarrow \mathcal{F}) \rightarrow \textbf{Method} \rightarrow (\mathcal{E}^{spec} \cup \mathcal{F}) \rightarrow \mathcal{F}$$

For calculating the wp^{src} predicate of an expression $\mathcal{E}^{src}$ declared in method `m`, the function wp^{src} takes as arguments $\mathcal{E}^{src}$, a postcondition nPost^{src}, an exceptional postcondition function $\mathsf{excPost}^{src}$ and returns the formula $wp^{src}(\mathcal{E}^{src}, \psi, \mathsf{excPost}^{src}, \texttt{m})_v$ which is the wp precondition of $\mathcal{E}^{src}$ if the its evaluation is represented by specification expression or formula v.

In Fig. 7.10 we can see the wp^{src} rules for the expressions of the source language except for method invokation and instance creation. The latter are given separately in Fig.7.11 as their definition is similar and is different from the rest of the wp rules for expressions.

Let us first look in Fig. 7.10. For instance, the rule for constant expressions does not change the state and thus, if a predicate nPost^{src} holds after its execution this means that it held in the prestate of the expession. Note that here we do not consider arithmetic expressions that may throw an arithmetic exception, i.e. we discard the division operations. However, we do not lose any particular feature of the language by discarding this case while gaining clearer representation. The rule for the $\mathcal{E}^{src}$ **instanceof** `Cl` expression is equal to the weakest precondition of the expression $\mathcal{E}^{src}$. This definition will allow us to get the possible outcomings of the evaluation of $\mathcal{E}^{src}$. Note that the value of the **instanceof** expression will be true if the evaluation of $\mathcal{E}^{src}$ is not **null** and is a subtype of `Cl`. The rest of the rules for relational expressions are defined in a similar way.

The rule for a cast expression (`Cl`) $\mathcal{E}^{src}$ have a similar semantics as the **instanceof** expression. However, its rule reflects the fact that the expression evaluation may terminate on a `CastExc` if $\mathcal{E}^{src}$ is not a subtype of `Cl`.

The rule for the field access expression takes into account the two possible outcomings of its evaluation. If the evaluation v of the expression $\mathcal{E}^{src}$ is different from **null** then the evaluation terminates normally, otherwise the exceptional postcondition for `NullPntrExc` must hold.

Let us now look at the rule for method invokation expression $\mathcal{E}^{src}$.`m`() in Fig. 7.11. The resulting precondition takes into account the case when the object v to which $\mathcal{E}^{src}$ evaluates and on which the method is called is **null** or not. If it is **null** then the weakest precondition of the handler against `NullPntrExc` must hold. In the opposite case, we want several predicates to hold. First, the precondition `m`.Pre^{src} of the invoked method `m` must hold. Then, the normal postcondition `m`.nPost^{src} of `m` must imply the postcondition nPost^{src} whatever is the value of the returned object and whatever are the values of the locations in the modifies list `m`.modif^{src} of the method `m`. Finally, if the method `m` terminates on an exception `E`, then the respective postcondition `m`.$\mathsf{excPostSpec}^{src}$(`E`) must imply the predicate $\mathsf{excPost}^{src}$(`E`) returned by the exceptional function $\mathsf{excPost}^{src}$ whatever are the values of the locations in the modifies list `m`.modif^{src} of the method `m`.

The rule for instance creation **new** `Cl`($\mathcal{E}^{src}$) is similar to method invokation rule. It states that for any reference which is fresh for the heap ($\neg$ **instances**(**bv**)) , which is not **null** and whose type is a subtype of `Cl` the precondition of the constructor constr(`Cl`) of class `Cl` must hold and that the normal postcondition constr(`Cl`).nPost^{src} of the constructor must imply the postcondition nPost^{src} with the respective substitutions. A similar condition we get for the cases when the constructor may terminate on an exception.

Statements

In the following, we present the rules of the weakest precondition predicate transformer for control statements. In Fig. 7.12 we give the rules for control statements which do not deal with exceptions. They are defined in a standard way. For instance, the rule seq states that the weakest predicate for a sequence of statements $\mathcal{STMT}_1; \mathcal{STMT}_2$ w.r.t. a normal postcondition nPost^{src} and exceptional postcondition function $\mathsf{excPost}^{src}$ is the weakest predicate of $\mathcal{STMT}_1$ w.r.t. to the normal postcondition $wp^{src}(\mathcal{STMT}_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})$ and the same exceptional postcondition function $\mathsf{excPost}^{src}$. Let us look also at the rule while. The rule requires that the invariant holds. It also requires that if the invariant holds and the conditional expression evaluates to true then the weakest precondition of the loop body w.r.t. a postcondition which is the loop invariant. Note that this is quantified over the expressions which are in the modifies list `modif` of the loop. This allows to properly initialize those variables which are not modified in the loop. Actually, we use the same technique here as in Chapter 5.

The control statements related to the exception handling and throwing as

$$wp^{src}(const, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_{const} = \mathsf{nPost}^{src}$$
$$const \in \{\mathbf{constInt}, true, false, \mathbf{constRef}, \mathbf{null}, \mathbf{this}, \mathbf{var}\}$$

$$wp^{src}(\mathcal{E}_1^{src}\ op\ \mathcal{E}_2^{src}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_{v_1 op v_2} =$$
$$wp^{src}(\mathcal{E}_1^{src}, wp^{src}(\mathcal{E}_2^{src}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_{v_2}, \mathsf{excPost}^{src}, \mathtt{m})_{v_1}$$
$$where\ op = \{+, -, *, \}$$

$$wp^{src}(\mathcal{E}^{src}\ \mathbf{instanceof}\ \mathtt{Cl}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_{v \neq \mathbf{null} \wedge \backslash \mathbf{typeof}(v) <: \mathtt{Cl}} =$$
$$wp^{src}(\mathcal{E}^{src}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_{v}$$

$$wp^{src}(\mathcal{E}_1^{src}\ \mathcal{R}\ \mathcal{E}_2^{src}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_{v_1 \mathcal{R} v_2} =$$
$$wp^{src}(\mathcal{E}_1^{src}, wp^{src}(\mathcal{E}_2^{src}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_{v_2}, \mathsf{excPost}^{src}, \mathtt{m})_{v_1}$$

$$wp^{src}(\mathcal{E}^{src}.f, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_{v.f} =$$
$$wp^{src}(\mathcal{E}^{src},$$
$$v \neq \mathbf{null} \Rightarrow \mathsf{nPost}^{src}$$
$$\wedge$$
$$v = \mathbf{null} \Rightarrow$$
$$\forall \mathbf{bv}, \neg \mathbf{instances}(\mathbf{bv}) \wedge \quad ,$$
$$\mathbf{bv} \neq \mathbf{null} \Rightarrow$$
$$\mathsf{excPost}^{src}(\ \mathtt{NullPntrExc})[\backslash \mathbf{EXC} \backslash \mathbf{bv}]$$
$$\mathsf{excPost}^{src}, \mathtt{m})_{v}$$

$$wp^{src}((\mathtt{Cl})\ \mathcal{E}^{src}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_{v} =$$
$$wp^{src}(\mathcal{E}^{src},$$
$$\backslash \mathbf{typeof}(v) <: \mathtt{Cl} \Rightarrow \mathsf{nPost}^{src}$$
$$\wedge$$
$$\neg\ \backslash \mathbf{typeof}(v) <: \mathtt{Cl} \Rightarrow$$
$$\forall \mathbf{bv}, \neg \mathbf{instances}(\mathbf{bv}) \wedge \quad ,$$
$$\mathbf{bv} \neq \mathbf{null} \Rightarrow$$
$$\mathsf{excPost}^{src}(\ \mathtt{CastExc})[\backslash \mathbf{EXC} \backslash \mathbf{bv}]$$
$$\mathsf{excPost}^{src}, \mathtt{m})_{v}$$

Figure 7.10: WP FOR SOURCE EXPRESSIONS

well as the finally statements may be have a more particular definition. They are given in Fig. 7.13. Let us look at the rule try catch. Actually, it is similar to the rule seq from Fig. 7.12, but dual in the way the postcondition modifications. In particular, the weakest predicate of a try catch statement $\mathtt{try}\ \{\mathcal{STMT}_1\}$ $\mathtt{catch}(\mathtt{Exc}\ c)\ \{\mathcal{STMT}_2\}$ w.r.t. a normal postcondition nPost^{src} and exceptional postcondition function $\mathsf{excPost}^{src}$ is the weakest predicate of the try statement $\mathcal{STMT}_1$ w.r.t. the normal postcondition nPost^{src} and the updated exceptional function $\mathsf{excPost}^{src}[\oplus \mathtt{Exc} \rightarrow wp^{src}(\mathcal{STMT}_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})]$.

7.4.3 Example

In the following, we return back to method square in Fig. 7.2 and we apply the *wp* function over a fragment of the method. We concentrate on the part of the code starting at the loop statement and we show the preconditions calculated w.r.t. the method specification in Fig. 7.14.

$$
\begin{array}{l}
wp^{src}(\mathcal{E}^{src}.\mathtt{m}(), \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_v = \\
\quad wp^{src}(\mathcal{E}^{src}, \\
\qquad \left\{ \begin{array}{l}
v' \neq \mathbf{null} \Rightarrow \\
\qquad \mathtt{m}.\mathsf{Pre}^{src}[\mathbf{this}\backslash v'] \\
\qquad \wedge \\
\qquad \forall \mathbf{bv},\ \forall\ m \in \mathtt{m}.\mathsf{modif}^{src} \\
\qquad \left\{ \begin{array}{l} \backslash\mathbf{typeof(bv)} <: \mathtt{m}.\mathsf{retType} \wedge \\ \mathtt{m}.\mathsf{nPost}^{src} \quad [\backslash\mathbf{result}\backslash\mathbf{bv}] \\ \qquad\qquad\quad [\mathbf{this}\backslash v'] \\ \qquad\quad \Rightarrow \mathsf{nPost}^{src}[v\backslash\mathbf{bv}] \end{array} \right. \\
\qquad \wedge \\
\qquad \forall \mathtt{E} \in \mathtt{m}.\mathsf{exceptions}^{src}, \\
\qquad \forall\ m \in \mathtt{m}.\mathsf{modif}^{src} \\
\qquad\ \forall \mathbf{bv}, \mathbf{bv} \neq \mathbf{null} \wedge \\
\qquad\ \backslash\mathbf{typeof(bv)} <: \mathtt{E} \Rightarrow \\
\qquad\ \mathtt{m}.\mathsf{excPostSpec}^{src}(\mathtt{E}) \Rightarrow \mathsf{excPost}^{src}(\mathtt{E})[\backslash\mathbf{EXC}\backslash\mathbf{bv}] \\
v' = \mathbf{null} \Rightarrow \\
\qquad \forall \mathbf{bv}, \neg\mathbf{instances}(\mathbf{bv}) \wedge \\
\qquad \mathbf{bv} \neq \mathbf{null} \wedge \\
\qquad \backslash\mathbf{typeof(bv)} <:\ \mathtt{NullPntrExc} \Rightarrow \\
\qquad\quad \mathsf{excPost}^{src}(\ \mathtt{NullPntrExc})[\backslash\mathbf{EXC}\backslash\mathbf{bv}]
\end{array} \right. , \\
\qquad \mathsf{excPost}^{src}, \mathtt{m})_{v'}
\end{array}
$$

$$
\begin{array}{l}
wp^{src}(\mathbf{new}\ \mathtt{Cl}(\mathcal{E}^{src}), \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_v = \\
\quad \forall \mathbf{bv}, \\
\quad \neg\ \mathbf{instances}(\mathbf{bv}) \wedge \\
\quad \mathbf{bv} \neq \mathbf{null} \\
\quad \backslash\mathbf{typeof(bv)} <: \mathtt{Cl} \Rightarrow \\
\quad wp^{src}(\mathcal{E}^{src}, \\
\qquad \left\{ \begin{array}{l}
\mathsf{constr}(\mathtt{Cl}).\mathsf{Pre}^{src} \ \begin{array}{l}[\mathbf{this}\backslash\mathbf{bv}] \\ [arg\backslash v'] \\ [f\backslash f[\oplus\mathbf{bv} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\mathsf{subtype}(f.\mathsf{declaredIn},\mathtt{Cl})}\end{array} \\
\wedge \\
\forall\ m \in \mathsf{constr}(\mathtt{Cl}).\mathsf{modif}^{src}, \\
\mathsf{constr}(\mathtt{Cl}).\mathsf{nPost}^{src} \ \begin{array}{l}[\mathbf{this}\backslash\mathbf{bv}] \\ [arg\backslash v'] \\ [f\backslash f[\oplus\mathbf{bv} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\mathsf{subtype}(f.\mathsf{declaredIn},\mathtt{Cl})}\end{array} \\
\qquad \Rightarrow \ \begin{array}{l}\mathsf{nPost}^{src} \\ [v\backslash\mathbf{bv}] \\ [f\backslash f[\oplus\mathbf{bv} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\mathsf{subtype}(f.\mathsf{declaredIn},\mathtt{Cl})}\end{array} \\
\wedge \\
\forall \mathtt{Exc} \in \mathsf{constr}(\mathtt{Cl}).\mathsf{exceptions}^{src}, \\
\forall\ m \in \mathsf{constr}(\mathtt{Cl}).\mathsf{modif}^{src}, \\
\forall \mathbf{bv}_{exc}, \\
\left\{ \begin{array}{l} \mathbf{bv} \neq \mathbf{null} \wedge \\ \backslash\mathbf{typeof}(\mathbf{bv}_{exc}) <: \mathtt{Exc} \Rightarrow \\ \mathsf{constr}(\mathtt{Cl}).\mathsf{excPostSpec}^{src}(\mathtt{Exc}) \Rightarrow \\ \qquad \mathsf{excPost}^{src}(\mathtt{Exc}) \end{array} \right\} \begin{array}{l}[\backslash\mathbf{EXC}\backslash\mathbf{bv}_{exc}] \\ [v\backslash\mathbf{bv}] \\ [f\backslash f[\oplus\mathbf{bv} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\mathsf{subtype}(f.\mathsf{declaredIn},\mathtt{Cl})}\end{array}
\end{array} \right. , \\
\qquad \mathsf{excPost}^{src}, \mathtt{m})_{v'}
\end{array}
$$

Figure 7.11: WEAKEST PRECONDITION FOR METHOD INVOKATION AND INSTANCE CREATION

7.5 Weakest precondition calculus for bytecode programs

In this section, we introduce a new formulation of the *wp* function for bytecode which will be based on the compiler from source to bytecode language. The

$$wp^{src}(\mathcal{STMT}_1; \mathcal{STMT}_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =^{\mathsf{seq}}$$
$$wp^{src}(\mathcal{STMT}_1, wp^{src}(\mathcal{STMT}_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}), \mathsf{excPost}^{src}, \mathtt{m})$$

$$wp^{src}(\mathcal{E}_1^{src} = \mathcal{E}_2^{src}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =^{\mathsf{locVarAssign}}$$
$$wp^{src}(\mathcal{E}_2^{src}, \mathsf{nPost}^{src}[\mathcal{E}_1^{src} \backslash v], \mathsf{excPost}^{src}, \mathtt{m})_v$$

$$wp^{src}(\mathcal{E}_1^{src}.f = \mathcal{E}_2^{src}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =^{\mathsf{fieldAssign}}$$
$$wp^{src}(\mathcal{E}_1^{src},$$
$$wp^{src}(\mathcal{E}_2^{src},$$
$$v_1 \neq \mathbf{null} \Rightarrow$$
$$\mathsf{nPost}^{src}[f \backslash f[\oplus v_1 \rightarrow v_2]]$$
$$\wedge \qquad ,$$
$$v_1 = \mathbf{null} \Rightarrow$$
$$\mathsf{excPost}^{src}(\mathtt{NullPointerExc})$$
$$\mathsf{excPost}^{src}, \mathtt{m})_{v_2},$$
$$\mathsf{excPost}^{src}, \mathtt{m})v_1$$

$$wp^{src}(\begin{array}{l}\mathtt{if}\ (\mathcal{E}^{\mathcal{R}}) \\ \mathtt{then}\{\mathcal{STMT}_1\} \\ \mathtt{else}\ \{\mathcal{STMT}_2\}\end{array}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =^{\mathsf{if}}$$
$$wp^{src}(\mathcal{E}^{\mathcal{R}},$$
$$v = true \Rightarrow wp^{src}(\mathcal{STMT}_1, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})$$
$$\wedge \qquad ,$$
$$v = false \Rightarrow wp^{src}(\mathcal{STMT}_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})$$
$$\mathsf{excPost}^{src}, \mathtt{m})_v$$

$$wp^{src}(\mathtt{while}\ (\mathcal{E}^{\mathcal{R}})\ [\mathtt{INV}, \mathtt{modif}]\ \ \{\mathcal{STMT}\}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =^{\mathsf{while}}$$
$$\mathtt{INV} \wedge$$
$$\forall\ m, m \in \mathtt{modif},$$
$$\mathtt{INV} \Rightarrow$$
$$wp^{src}(\mathcal{E}^{\mathcal{R}},$$
$$v = true \Rightarrow \ wp^{src}(\mathcal{STMT}, \mathtt{INV}, \mathsf{excPost}^{src}, \mathtt{m}) \qquad ,$$
$$v = false \Rightarrow \mathsf{nPost}^{src}$$
$$\mathsf{excPost}^{src}, \mathtt{m})_v$$

$$wp^{src}(\mathtt{return}\ \mathcal{E}^{src}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =$$
$$wp^{src}(\mathcal{E}^{src}, \mathsf{nPost}^{src}[\backslash \mathbf{result} \backslash v], \mathsf{excPost}^{src}, \mathtt{m})_v$$

Figure 7.12: WP FOR SOURCE CONTROL STATEMENTS WITHOUT EXCEPTIONS

motivation for this new definition is that it will allow to reason about the relation between source and bytecode proof obligations. Of course, it is also important to see what is the relation between the new definition of the *wp* introduced here and the definition given earlier in Chapter 5, section 5.3. We will argue under what conditions the two formulations of the *wp* function produce the same formulas.

We give now a definition of the *wp* function for a single sequential instruction (instructions different from goto, if_cond) which takes explicitely the postcondition and the exceptional postcondition function upon which the precondition will be calculated. Its signature is the following:

$$
\begin{array}{l}
wp^{src}(\texttt{throw}\ \mathcal{E}^{src}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \texttt{m}) =^{\mathsf{throw}} \\
\quad wp^{src}(\mathcal{E}^{src}, \\
\qquad v = \textbf{null} \Rightarrow \mathsf{excPost}^{src}(\texttt{NullPointerExc}) \\
\qquad \wedge \\
\qquad v \neq \textbf{null} \Rightarrow \\
\qquad\quad \forall \texttt{Exc}\ , \\
\qquad\quad \backslash\textbf{typeof}(v) <: \texttt{Exc}\ \Rightarrow \\
\qquad\quad\quad \texttt{m}.\mathsf{excPost}^{src}(\texttt{Exc}\ )[\backslash\textbf{EXC}\backslash v] \\
\qquad \mathsf{excPost}^{src}, \texttt{m})_v
\end{array}
,
$$

$$
\begin{array}{l}
wp^{src}(\texttt{try}\ \{\mathcal{STMT}_1\}\ \texttt{catch(Exc}\ c\texttt{)}\ \{\mathcal{STMT}_2\}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \texttt{m}) =^{\mathsf{try\ catch}} \\
\quad wp^{src}(\mathcal{STMT}_1, \\
\qquad \mathsf{nPost}^{src}, \\
\qquad \mathsf{excPost}^{src}[\oplus\texttt{Exc} \rightarrow wp^{src}(\mathcal{STMT}_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \texttt{m})], \texttt{m})
\end{array}
$$

$$
\begin{array}{l}
wp^{src}(\texttt{try}\ \{\mathcal{STMT}_1\}\ \texttt{finally}\ \{\mathcal{STMT}_2\}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \texttt{m}) =^{\mathsf{try\ finally}} \\
\quad wp^{src}(\mathcal{STMT}_1, \\
\qquad wp^{src}(\mathcal{STMT}_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \texttt{m}), \\
\qquad \mathsf{excPost}^{src}[\oplus\texttt{Exception} \rightarrow wp^{src}(\mathcal{STMT}_2, \mathsf{excPost}^{src}(\texttt{Exception}), \mathsf{excPost}^{src}, \texttt{m})], \texttt{m})
\end{array}
$$

Figure 7.13: WP FOR THE CONTROL STATEMENTS WITH EXCEPTIONS

```
1  {forall s, sqr,
2         (0 <= s && s <= v && sqr == s*s )&& (s<v) ==> Prebody
3  &&
4  forall s, sqr,
5         (0 <= s && s <= v && sqr == s*s )&& (s>=v) ==> sqr = i*i
6  &&
7  0 <= s && s <= v && sqr == s*s}
8  while( s < v ) {
9
10   {PreBody: 0 <= s+1 && s+1 <= v && sqr + 2*s + 1 == (s+1)*(s+1)}
11   sqr = sqr + 2*s + 1;
12
13   {0 <= s+1 && s+1 <= v && sqr == (s+1)*(s+1)}
14   s = s+1;}
15
16 {sqr = i*i }
17 return sqr;
18
19 {Post: \result = i*i}
```

Figure 7.14: METHOD SQUARE WRITTEN IN OUR SOURCE LANGUAGE

$$\mathrm{wp}^{bc} : \ \mathrm{I}\backslash\{\text{goto}, \text{if\_cond}\} \rightarrow P \rightarrow (\ \texttt{ExcType} \rightarrow P) \rightarrow P$$

For instance, the *wp* definition for getfield is :

$$
\begin{array}{l}
wp^{bc}(\text{getfield}\ f, \psi, \mathsf{excPost}^{src2bc}, \texttt{m}) = \\
\textbf{st}(\textbf{cntr}) \neq \textbf{null} \Rightarrow \psi[\textbf{st}(\textbf{cntr})\backslash f(\textbf{st}(\textbf{cntr}))] \wedge \\
\textbf{st}(\textbf{cntr}) = \textbf{null} \Rightarrow \mathsf{excPost}^{src2bc}(\ \texttt{NullPntrExc})
\end{array}
$$

Note that this differs from the definition of the *wp* given in Chapter 5, section 5.3 where the postcondition is a function of the successor of the current instruction. We do not give the rest of the rules because they are the same as the rules presented in 5.3 except for the fact that the local postconditions are given explicitely.

We also define the weakest predicate transformer function for a block of instructions as follows:

Definition 7.5.1 (*wp* for a block of instructions)

$$wp^{bc}_{seq}(1;\ldots;k,\psi,\mathsf{excPost}^{src2bc},\mathtt{m}) =$$
$$wp^{bc}_{seq}(1;\ldots;k-1,wp^{bc}(k,\psi,\mathsf{excPost}^{src2bc},\mathtt{m}),\mathsf{excPost}^{src2bc},\mathtt{m})$$

We turn now to the rules for compiled expressions. Note that from Property 7.3.4.3 it follows that the compilation of any expression is a sequence of instructions that execute sequentially and there is no jump from outside inside the sequence. Thus, we use the predicate transformer for a sequence of bytecode instructions defined above in order to define the predicate transformer for expressions.

Definition 7.5.2 (*wp* for compiled expressions) *For any expression $\mathcal{E}^{src}$, postcondition ψ and exceptional postcondition function* $\mathsf{excPost}^{src2bc}$ *the wp function for the compilation $\ulcorner s,\mathcal{E}^{src},e\urcorner_{\mathtt{m}}$ is $wp^{bc}_{seq}(\ulcorner s,\mathcal{E}^{src},e\urcorner_{\mathtt{m}},\psi,\mathsf{excPost}^{src2bc},\mathtt{m})$*

For instance, the rule of the *wp* for the compilation of access field expression $\mathcal{E}^{src}.f$ where its compilation is

$$\begin{array}{l}\ulcorner s,\mathcal{E}^{src}_1;,e-1\urcorner_{\mathtt{m}}\\ e \text{ getfield } f\end{array}$$

produce the following formula

$$wp^{bc}_{seq}(\begin{array}{l}\ulcorner s,\mathcal{E}^{src},e-1\urcorner_{\mathtt{m}};\\ e \text{ getfield } f\end{array},\psi,\mathsf{excPost}^{src2bc},\mathtt{m})$$

This is equivalent to

$$wp^{bc}_{seq}(\ulcorner s,\mathcal{E}^{src},e-1\urcorner_{\mathtt{m}},wp^{bc}(\text{getfield } f,\psi,\mathsf{excPost}^{src2bc},\mathtt{m}),\mathsf{excPost}^{src2bc},\mathtt{m})$$

The function which calculates the *wp* predicate of a compiled statement is called wp^{bc}_{stmt} and has the following signature :

$$wp^{bc}_{stmt} : Set(\ \mathrm{I}) \to P \to (\ \mathtt{Exc} \to P) \to P$$

The definition of wp^{bc}_{stmt} is shown in Fig. 7.15

As we can see in the figure the *wp* is defined over the result of the compiler for every statement. Thus, it is similar to the definition of weakest precondition function for the source language.

7.5.1 Properties of the *wp* functions

The previous subsection introduced a new formulation of the *wp* function for bytecode which is defined over the source statement from which it is compiled. However, it is important to establish a relation between this new definition and the *wp* formulation given in Chapter 5. The following statements establish the relation between the two versions of the *wp* calculus.

The first lemma states that the wp^{bc} function defined in the previous subsection for a single bytecode instruction will return the same result as the *wp* function defined in Chapter 5. In particular, the explicite weakest precondition function for a single bytecode instruction $wp^{bc}(i_j, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})$ is equivalent to $wp(i_j\ , \mathtt{m})$ only if ψ is the intermediate predicate between j and the next instruction k to be executed. A similar condition we get for the function $\mathsf{excPost}^{src2bc}$.

Lemma 7.5.1.1 (Equivalence of the formulations for single instructions) *For all instructions* $i_j \in$ I\{goto, if_cond} *and* i_k *which belong to method* m*, formula* ψ*,and function* $\mathsf{excPost}^{src2bc}$: ExcType $\rightarrow P$ *such that*

- $i_j \rightarrow i_k$
- $\psi =$ $inter(i_j, i_k, \mathtt{m})$
- $\forall \mathtt{Exc}, \mathsf{excPost}^{src2bc}(\mathtt{Exc}) = \mathtt{m}.\mathsf{excPostIns}(\mathtt{Exc}, j)$

the following holds
$wp^{bc}(i_j, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = wp(i_j\ , \mathtt{m})$

The proof is done by case analysis on the instruction i_j

Proof: We scatch the case for a getfield instruction.

{ *by hypothesis* }
$i_j =$ getfield f
{ *by definition of the wp function* }
(1) wp^{bc}(getfield $f, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) =$
$\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow \psi[\mathbf{st}(\mathbf{cntr})\backslash f(\mathbf{st}(\mathbf{cntr}))]$
$\wedge$
$\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathsf{excPost}^{src2bc}(\ \mathtt{NullPntrExc})$

{ *by definition of the wp function* }
(2) wp(getfield $f\ , \mathtt{m}) =$
$\mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow$ $inter(i_j, i_k, \mathtt{m})$ $[\mathbf{st}(\mathbf{cntr})\backslash f(\mathbf{st}(\mathbf{cntr}))]$
$\wedge$
$\mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathsf{excPostRTE}(\ \mathtt{NullPntrExc}, j)$

{ *from the initial hypothesis, (1) and (2) the lemma holds in that case* }

Qed.

The following lemma establishes that calculating the *wp* predicate of the first instruction of a block of bytecode instructions and calculating it by using a wp^{bc}_{seq} is the same under several conditions about the postcondition predicates.

Lemma 7.5.1.2 (*wp* for a block of instructions) *For every block of instructions* $1;\ldots;j$ *and instruction* k *in method* $\mathtt{m}$*, formula* ψ *and function* $\mathsf{excPost}^{src2bc}$: $\mathtt{ExcType} \rightarrow P$ *such that*

- $j \rightarrow k$
- $\psi = \ inter(j, j+1, \mathtt{m})$
- $\forall \mathtt{Exc}, \forall i, 1 \leq i \leq j, \mathsf{excPost}^{src2bc}(\mathtt{Exc}) = \mathtt{m}.\mathsf{excPostIns}(\mathtt{Exc}, i)$

then the following holds

$$wp^{bc}_{seq}(1;\ldots;j, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = wp(1\ , \mathtt{m})$$

The proof is done by induction on the length of the sequence of instructions.
Proof:

(1) $wp^{bc}_{seq}(1;\ldots;j, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) =$
{ *by Def. 7.5.1 for wp of block of instructions* }
$wp^{bc}_{seq}(1;\ldots;j-1, wp^{bc}(j, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}), \mathsf{excPost}^{src2bc}, \mathtt{m})$
{ *by initial hypothesis we can apply lemma 7.5.1.1 from which it follows* }
$wp^{bc}(j, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = wp(\ j, \mathtt{m})$
{*by definition 7.3.4.1 , j is not a loop entry and from Def. 5.3.1* }
(2) $inter(j, j+1, \mathtt{m}) = wp(\ j, \mathtt{m})$
{*apply the induction hypothesis over* $1;\ldots;j-1$ *, (2) and the initial hypothesis for exception handlers* }
(3) $wp^{bc}_{seq}(1;\ldots;j-1, wp^{bc}(j, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}), \mathsf{excPost}^{src2bc}, \mathtt{m}) =$
$wp(\ 1, \mathtt{m})$
{ *from (1) and (3) the proposition holds*}

Qed.

The same property is established for compilation of expressions.

Lemma 7.5.1.3 (*wp* for compiled expressions) *For every compiled expression* $\ulcorner s, \mathcal{E}^{src}, e \urcorner_{\mathtt{m}}$ *in method* $\mathtt{m}$ *formula* ψ *and function* $\mathsf{excPost}^{src2bc}$: $\mathtt{ExcType} \rightarrow P$ *such that*

- $\psi = \ inter(e, e+1, \mathtt{m})$
- $\forall \mathtt{Exc}, \forall 1 \leq i, k \leq j, \mathsf{excPost}^{src2bc}(\mathtt{Exc}) = \mathtt{m}.\mathsf{excPostIns}(\mathtt{Exc}, i)$

then the following holds:

$$wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{src}, e \urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = wp(s\ , \mathtt{m})$$

Proof:
From Property 7.3.4.6 of the compiler it follows that for every expression $\mathcal{E}^{src}$, start label s and end label e, the resulting compilation $\ulcorner s, \mathcal{E}^{src}, e \urcorner_{\mathtt{m}}$ is a block of instructions. We can apply the previous lemma 7.5.1.2 and we get the result. *Qed.*

The next lemma states the same property but this time for the compilation of statements.

Lemma 7.5.1.4 *For every compiled statement* $\ulcorner s, \mathcal{STMT}, e \urcorner_{\mathtt{m}}$ *in method* $\mathtt{m}$*, formula* ψ *and function* $\mathsf{excPost}^{src2bc} : \mathtt{ExcType} \rightarrow P$ *such that*

- $\psi = \ inter(e, e+1, \mathtt{m})$
- $\forall \mathtt{Exc}, \mathsf{excPost}^{src2bc}(\mathtt{Exc}) = \mathtt{m}.\mathsf{excPostIns}(\mathtt{Exc}, e)$

then the following holds:

$$wp^{bc}_{stmt}(\ulcorner s, \mathcal{STMT}, e \urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = wp(s\ , \mathtt{m})$$

Proof : the proof is by induction on the compilation of a statement. We scatch here the proof of few cases

- if statement

$$wp^{bc}_{stmt}(\ulcorner s,\ \begin{array}{l}\texttt{if } (\mathcal{E}^{\mathcal{R}})\\ \texttt{then } \{\mathcal{ST}\mathcal{MT}_1\}\\ \texttt{else } \{\mathcal{ST}\mathcal{MT}_2\}\end{array}\ , e\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) =^{def}$$

$$wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{\mathcal{R}}, e'\urcorner_{\mathtt{m}};$$
,
$$\begin{array}{l}\mathcal{R}(\mathbf{st}(\mathbf{cntr}), \mathbf{st}(\mathbf{cntr}-1)) \Rightarrow\\ \qquad wp^{bc}_{stmt}(\ulcorner e''+2, \mathcal{ST}\mathcal{MT}_1, e\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})[t\backslash t-2]\\ \wedge\\ \neg\mathcal{R}(\mathbf{st}(\mathbf{cntr}), \mathbf{st}(\mathbf{cntr}-1)) \Rightarrow\\ \qquad wp^{bc}_{stmt}(\ulcorner e'+2, \mathcal{ST}\mathcal{MT}_2, e''\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})[t\backslash t-2]\\ \mathsf{excPost}^{src2bc}, \mathtt{m})\end{array}\ ,$$

{ by initial hypothesis, we have }
(1.1) $\psi = inter(e, e+1, \mathtt{m})$
{ by the compiler definition $e''+1 =$ goto $e+1$; }
(1.2) $e''+1 \rightarrow e+1$
{ *from Property 7.3.4.8 which states that every instruction inside the compilation of* $\ulcorner s, \mathcal{ST}\mathcal{MT}, e\urcorner_{\mathtt{m}}$ *, is in the same execution relation with* $e+1$*, Def. 5.3.1.1 for inter, (1.1) and (1.2) we conclude that* }
(1.3) $inter(e''+1, e+1, \mathtt{m}) = inter(e, e+1, \mathtt{m}) = \psi$
{*From property 7.3.4.7 the edge between* e'' *and* $e''+1$ *is not a loop edge and from Def. 5.3.1.1* }
(1.4) $inter(e'', e''+1, \mathtt{m}) = wp(\ e''+1, m)$
{ *From the definition of the wp for* goto *instructions and (1.4)* }
(1.5) $wp(\ e''+1, m) = inter(e''+1, e+1, \mathtt{m})$
{ *From (1.4), (1.3) and (1.5)* }
(1.6) $inter(e'', e''+1, \mathtt{m}) = inter(e, e+1, \mathtt{m}) = \psi$
{ *From Property 7.3.4.10* }

(1.7) $\mathtt{m.excPostIns(Exc}, e) = \mathtt{m.excPostIns(Exc}, e'')$
{ *apply the induction hypothesis over (1.6) and (1.7)* }
(1.8) $wp^{bc}_{stmt}(\ulcorner e'+2, \mathcal{STMT}_2, e''\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = wp(\ e'+2, m)$
{ *apply the induction hypothesis over the initial hypothesis* }
(1.9) $wp^{bc}_{stmt}(\ulcorner e''+2, \mathcal{STMT}_1, e\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = wp(\ e''+2, m)$
{ *by definition of* if_cond *function* }
(1.10) $wp(\ e'+1 : \text{if\_cond}\ e''+2;, m) =$
$\mathcal{R}(\mathbf{st}(\mathbf{cntr}), \mathbf{st}(\mathbf{cntr}-1)) \Rightarrow wp(\ e''+2, m)$
$\wedge$
$\neg\mathcal{R}(\mathbf{st}(\mathbf{cntr}), \mathbf{st}(\mathbf{cntr}-1)) \Rightarrow wp(\ e'+2, m)$
{ *from (1.8), (1.9) and (1.10)*}
(1.11) $wp(\ e'+1 : \text{if\_cond}\ e''+2;, m) =$
$\mathcal{R}(\mathbf{st}(\mathbf{cntr}), \mathbf{st}(\mathbf{cntr}-1)) \Rightarrow$
$wp^{bc}_{stmt}(\ulcorner e''+2, \mathcal{STMT}_1, e\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})$
$\wedge$
$\neg\mathcal{R}(\mathbf{st}(\mathbf{cntr}), \mathbf{st}(\mathbf{cntr}-1)) \Rightarrow$
$wp^{bc}_{stmt}(\ulcorner e'+2, \mathcal{STMT}_2, e''\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})$
{ *From Property 7.3.4.8* }
(1.12) $inter(e', e'+1, \mathtt{m}) = wp(\ e'+1 : \text{if\_cond}\ e''+2;, m)$
{ *From the initial hypothesis* }
(1.13) $\mathtt{m.excPostIns(Exc}, e) = \mathsf{excPost}^{src2bc}(\mathtt{Exc})$
{ *From Property 7.3.4.11 and then (1.13)*}
(1.14) $\forall \mathtt{Exc}, \forall i, s \leq i \leq e', \mathsf{excPost}^{src2bc}(\mathtt{Exc}) =$
$\mathtt{m.excPostIns(Exc}, i)$
{ *apply Lemma 7.5.1.3 over hypothesis (1.11), (1.12), (1.14)* }
$wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{\mathcal{R}}, e'\urcorner_{\mathtt{m}};$
,
$\mathcal{R}(\mathbf{st}(\mathbf{cntr}), \mathbf{st}(\mathbf{cntr}-1)) \Rightarrow$
$wp^{bc}_{stmt}(\ulcorner e''+2, \mathcal{STMT}_1, e\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})[t\backslash t-2]$
$\wedge$,
$\neg\mathcal{R}(\mathbf{st}(\mathbf{cntr}), \mathbf{st}(\mathbf{cntr}-1)) \Rightarrow$
$wp^{bc}_{stmt}(\ulcorner e'+2, \mathcal{STMT}_2, e''\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})[t\backslash t-2]$
$\mathsf{excPost}^{src2bc}, \mathtt{m}) = wp(\ s, m)$

{ *and this case holds*}

- try catch statement

$$
\begin{array}{l}
\textit{(1) } wp^{bc}_{stmt}(\ulcorner s,\ \begin{array}{l}\texttt{try}\ \{\mathcal{STMT}_1\}\\ \texttt{catch}\ (\mathit{ExcType}\ \textbf{var})\{\mathcal{STMT}_2\}\end{array}\ , e\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) =^{def}\\
\{\ \textit{Def. of } wp^{bc}_{stmt}\ \textit{in the previous Section } \textbf{??}\ \}\\
wp^{bc}_{stmt}(\ulcorner s, \mathcal{STMT}_1, e'\urcorner_{\mathtt{m}};,\\
\qquad \psi,\\
\qquad \mathsf{excPost}^{src2bc}[\oplus \mathit{ExcType} \rightarrow wp^{bc}_{stmt}(\ulcorner e'+2, \mathcal{STMT}_2, e\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})], \mathtt{m})\\
\{\ \textit{apply the induction hypothesis over } \mathcal{STMT}_2 \textit{ and the initial hypothesis}\}\\
\textit{(2) } wp^{bc}_{stmt}(\ulcorner e'+2, \mathcal{STMT}_2, e\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = wp(\ e'+2, \mathtt{m})\\
\{\ \textit{from the definition of } \mathsf{excPostIns} \textit{ and property 7.3.4.12}\}\\
\textit{(3) } \forall \mathtt{Exc}, \mathsf{excPost}^{src2bc}[\oplus \mathit{ExcType} \rightarrow wp(\ e'+2, \mathtt{m})] =\\
\qquad \mathtt{m}.\mathsf{excPostIns}(\mathtt{Exc}, e')\\
\{\ \textit{we can also conclude from Prop. 7.3.4.8 and 7.3.4.7}\ \}\\
\textit{(4) } \mathit{inter}(e', e'+1, \mathtt{m}) = \mathit{inter}(e'+1, e+1, \mathtt{m}) = \mathit{inter}(e, e+1, \mathtt{m})\\
\{\ \textit{apply induction hypothesis over (1), (3) and (4)}\ \}\\
wp^{bc}_{stmt}(\ulcorner s, \mathcal{STMT}_1, e'\urcorner_{\mathtt{m}};,\\
\qquad \psi,\\
\qquad \mathsf{excPost}^{src2bc}[\oplus \mathit{ExcType} \rightarrow wp^{bc}_{stmt}(\ulcorner e'+2, \mathcal{STMT}_2, e\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})], \mathtt{m}) =\\
wp(\ s, \mathtt{m})
\end{array}
$$

Qed.

7.6 Proof obligation equivalence on source and bytecode level

In the following, we will argue that the proof obligations generated by the weakest precondition over programs of our source language are equivalent to the proof obligations generated over bytecode.

First, we remark that a wp^{src} defined over source expressions generates formulas which may have two possible normal forms.

Lemma 7.6.1 (Normal form of wp^{src}) *for any source expression $\mathcal{E}^{src}$ normal form of wp^{src} is such that*

$$
\begin{array}{l}
\textit{either there exists } Q, R : \mathcal{F} \\
wp^{src}(\mathcal{E}^{src}, \psi, \mathsf{excPostRTE}, \mathtt{m})_v \ = \\
Q \Rightarrow \psi \\
\wedge \\
R \\
\\
\textit{or exists } Q, R : \mathcal{F}, v_1 \ldots v_k : \mathcal{E}^{src} \\
\\
wp^{src}(\mathcal{E}^{src}, \psi, \mathsf{excPostRTE}, \mathtt{m})_v \ = \\
\qquad\qquad\qquad\qquad\qquad\qquad [v \backslash \mathbf{bv}] \\
\forall \mathbf{bv}, \mathbf{bv}_1, \ldots \mathbf{bv}_k, Q \Rightarrow \psi \quad [v_1 \backslash \mathbf{bv}_1] \quad \wedge \\
\qquad\qquad\qquad\qquad\qquad\qquad \ldots \\
\qquad\qquad\qquad\qquad\qquad\qquad [v_k \backslash \mathbf{bv}_k] \\
R
\end{array}
$$

This follows from the definition of wp^{src} in Section 7.4.2. The second form in which the part of the calculated weakest precondition predicate is quantified appears in the cases for instance creation and method invokation expressions.

We now turn to see what is the relation between the weakest precondition formulas for an expression $\mathcal{E}^{src}$ and its compilation $\ulcorner s, \mathcal{E}^{src}, e \urcorner_{\mathtt{m}}$. Depending on the form of the weakest precondition (as we saw with the previous lemma there are two possible forms of the weakest precondition for source expressions) of a soure expression $\mathcal{E}^{src}$ obtained from wp^{src} we will state the relation between wp^{src} and wp^{bc}_{seq} in the following two lemmas. Informally, both lemmas express the fact that

- the hypothesis of the part of the weakest preconditions on source and bytecode level concerning the normal termination of the expression are the same

- the part of the weakest preconditions on source and bytecode level concerning the exceptional termination of the expression are the same

Lemma 7.6.2 (Wp of a compiled expression) *For any expression $\mathcal{E}^{src}$ from our source language, for any formula $\psi : \mathcal{F}$ of the source assertion language and any formula $\phi : P$ such that ϕ may only contain stack expressions of the form $\mathbf{st}(\mathbf{cntr} - k), k \geq 0$, the following holds*

There exist $Q, R : \mathcal{F}$ *such that*

$$
\begin{array}{l}
wp^{src}(\mathcal{E}^{src}, \psi, \mathsf{excPostRTE}, \mathtt{m})_v \ = \\
Q \Rightarrow \psi \\
\wedge \\
R \\
\\
\Longrightarrow \\
\\
wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{src}, e \urcorner_{\mathtt{m}}, \phi, \mathsf{excPostRTE}, \mathtt{m}) \ = \\
Q \Rightarrow \phi \ \begin{array}{l}[\mathbf{cntr} \backslash \mathbf{cntr} + 1] \\ [\mathbf{st}(\mathbf{cntr} + 1) \backslash v]\end{array} \\
\\
\wedge \\
R
\end{array}
$$

The following lemma refers to the cases for method invokation and instance creation. In the following, we scatch the case for instance creation but we ommit the part of the formulas concerning the exceptional termination of the evaluation of the expression.

Lemma 7.6.3 (Wp of a compiled expression) *For any expression* $\mathcal{E}^{src}$ *from our source language, for any formula* $\psi : \mathcal{F}$ *of the source assertion language and any formula* $\phi : P$ *such that* ϕ *may only contain stack expressions of the form* $\mathbf{st}(\mathbf{cntr} - k), k \geq 0$ *and if the modifies clauses of every method does not contain stack expressions, then the following holds :*
there exist $Q, R : \mathcal{F}, v_1 \ldots v_k : \mathcal{E}^{src}$

actually, here need an assumption that the modifies clauses cannot contain stack expressions

$$
\begin{array}{l}
wp^{src}(\mathcal{E}^{src}, \psi, \mathsf{excPostRTE}, \mathtt{m})_v = \\
\forall \mathbf{bv}, \mathbf{bv}_1, \ldots \mathbf{bv}_k, Q \Rightarrow \\
\qquad \psi \ \begin{array}{l}[v \backslash \mathbf{bv}] \\ [v_1 \backslash \mathbf{bv}_1] \\ \ldots \\ [v_k \backslash \mathbf{bv}_k]\end{array} \ \wedge \\
R \\
\Longrightarrow \\
\\
wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{src}, e \urcorner_{\mathtt{m}}, \phi, \mathsf{excPostRTE}, \mathtt{m}) \ = \\
\forall \mathbf{bv}, \mathbf{bv}_1, \ldots \mathbf{bv}_k, Q \Rightarrow \\
\qquad \phi \ \begin{array}{l}[\mathbf{cntr} \backslash \mathbf{cntr} + 1] \\ [\mathbf{st}(\mathbf{cntr} + 1) \backslash \mathbf{bv}] \\ [v_1 \backslash \mathbf{bv}_1] \\ \ldots \\ [v_k \backslash \mathbf{bv}_k]\end{array} \\
\\
\wedge \\
R
\end{array}
$$

The next lemma establishes a relation between the source and bytecode functions *wp* w.r.t. to the same weakest precondition.

say here that we are aware that the names on bytecode and source level will not be the same. However this is a minor detail

Lemma 7.6.4 (Proof obligation equivalence on statements)

$$\begin{array}{l}\forall \mathcal{STMT}, \psi, \mathsf{excPost}^{src},\\ \quad wp^{bc}_{stmt}(\ulcorner s, \mathcal{STMT}, e\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src}, \mathtt{m}) =\\ \quad wp^{src}(\mathcal{STMT}, \psi, \mathsf{excPost}^{src}, \mathtt{m})\end{array}$$

This follows from the previous lemma (which establishes the relation between the *wp* over source and the *wp* over bytecode that takes into account the compilation structure) and lemma 7.5.1.4 (which establishes the relation between the *wp* over bytecode that takes into account the compilation structure and the original *wp*which does not consider the properties of the compiler)

Lemma 7.6.5 *For every $\mathcal{STMT}$, compilation $\ulcorner s, \mathcal{STMT}, e\urcorner_{\mathtt{m}}$ $wp(s\ , \mathtt{m})$ does not contain stack expressions*

The next statement establishes under what conditions the *wp* over source and the *wp* defined in the previous Chapter are equivalent.

Lemma 7.6.6 (Proof obligation equivalence on statements) *For every $\mathcal{STMT}$, compilation $\ulcorner s, \mathcal{STMT}, e\urcorner_{\mathtt{m}}$, formula ψ and exceptional postcondition function $\mathsf{excPost}^{src2bc}$ such that :*

- $\psi = \ inter(e, e+1, \mathtt{m})$
- $\forall \mathtt{Exc}, \mathsf{excPost}^{src2bc}(\mathtt{Exc}) = \mathtt{m}.\mathsf{excPostIns}(\mathtt{Exc}, e)$

then it holds

$$wp^{src}(\mathcal{STMT}, \psi, \mathsf{excPost}^{src}, \mathtt{m}) = wp(s\ , \mathtt{m})$$

Proof:
This follows from the previous Lemma 7.6.4 and Lemma 7.5.1.4
Qed.

From the last lemma follows that if the body of the method $\mathtt{m}$ is $\mathcal{STMT}$, and its compilation is $\ulcorner s, \mathcal{STMT}, e\urcorner_{\mathtt{m}}$, then the proof obligations generated over $\mathcal{STMT}$ upon postcondition m .normalPost and exceptional postcondition function m .excPostSpec and its compilation $\ulcorner s, \mathcal{STMT}, e\urcorner_{\mathtt{m}}$ are the same. The last theorem formalizes this fact.

Theorem 7.6.1 (Proof obligation preservation) *For all method $\mathtt{m}$ if its body is $\mathcal{STMT}$ such that its compilation is $\ulcorner s, \mathcal{STMT}, e\urcorner_{\mathtt{m}}$ we have the following:*

$$wp^{src}(\mathcal{STMT}, \mathtt{m}.\mathsf{normalPost}, \mathtt{m}.\mathsf{excPost}^{src}, \mathtt{m}) = wp(s\ , \mathtt{m})$$

$$
\begin{array}{l}
wp^{bc}_{stmt}(\ulcorner s, \mathcal{STMT}_1; \mathcal{STMT}_2, e\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = \\
\quad wp^{bc}_{stmt}(\ulcorner s, \mathcal{STMT}_1, e'\urcorner_{\mathtt{m}}; , \\
\qquad\quad wp^{bc}_{stmt}(\ulcorner e'+1, \mathcal{STMT}_2, s\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}), \\
\qquad\quad \mathsf{excPost}^{src2bc}, \mathtt{m})
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\quad \mathtt{if}\ (\mathcal{E}^{\mathcal{R}}) \\
wp^{bc}_{stmt}(\ulcorner s, \quad \mathtt{then}\ \{\mathcal{STMT}_1\} \quad , e\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = \\
\qquad\quad \mathtt{else}\ \{\mathcal{STMT}_2\} \\
\quad wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{\mathcal{R}}, e'\urcorner_{\mathtt{m}}; , \\
\qquad\quad \mathcal{R}(\mathbf{st}(\mathbf{cntr}), \mathbf{st}(\mathbf{cntr}-1)) \Rightarrow \\
\qquad\qquad\quad wp^{bc}_{stmt}(\ulcorner e''+2, \mathcal{STMT}_1, e\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})[t\backslash t-2] \\
\qquad\quad \wedge \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , \\
\qquad\quad \neg\mathcal{R}(\mathbf{st}(\mathbf{cntr}), \mathbf{st}(\mathbf{cntr}-1)) \Rightarrow \\
\qquad\qquad\quad wp^{bc}_{stmt}(\ulcorner e'+2, \mathcal{STMT}_2, e''\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})[t\backslash t-2] \\
\qquad\quad \mathsf{excPost}^{src2bc}, \mathtt{m})
\end{array}
$$

$$
\begin{array}{l}
wp^{bc}_{stmt}(\ulcorner s, \mathcal{E}^{src}_1.f = \mathcal{E}^{src}_2, e\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = \\
\quad wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{src}_1, e'\urcorner_{\mathtt{m}}, \\
\qquad\quad wp^{bc}_{seq}(\ulcorner e'+1, \mathcal{E}^{src}_2, e-1\urcorner_{\mathtt{m}}; \\
\qquad\quad , \\
\qquad\qquad\quad wp^{bc}(e\ putfield\ f, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}), \quad , \\
\qquad\qquad\quad \mathsf{excPost}^{src2bc}, \mathtt{m}) \\
\qquad\quad \mathsf{excPost}^{src2bc}, \mathtt{m})
\end{array}
$$

$$
\begin{array}{l}
wp^{bc}_{stmt}(\ulcorner s, \quad \mathtt{try}\ \{\mathcal{STMT}_1\} \\
\qquad\quad \mathtt{catch}\ (ExcType\ \mathbf{var})\{\mathcal{STMT}_2\} \quad , e\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = \\
\quad wp^{bc}_{stmt}(\ulcorner s, \mathcal{STMT}_1, e'\urcorner_{\mathtt{m}}; , \\
\qquad\quad \psi, \\
\qquad\quad \mathsf{excPost}^{src2bc}[\oplus ExcType \rightarrow wp^{bc}_{stmt}(\ulcorner e'+2, \mathcal{STMT}_2, e\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})], \mathtt{m})
\end{array}
$$

$$
\begin{array}{l}
wp^{bc}_{stmt}(\ulcorner s, \quad \mathtt{try}\ \{\mathcal{STMT}_1\} \\
\qquad\quad \mathtt{finally}\ \{\mathcal{STMT}_2\} \quad , e\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = \\
\quad wp^{bc}_{stmt}(\ulcorner s, \mathcal{STMT}_1, e'\urcorner_{\mathtt{m}} \\
\quad , \\
\qquad\quad wp^{bc}_{stmt}(\ulcorner e'+1, \mathcal{STMT}_2, e''\urcorner_{\mathtt{m}}; , \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}), \\
\qquad\quad \mathsf{excPost}^{src2bc}[\oplus ExcType \rightarrow wp^{bc}_{stmt}(\begin{array}{l} e''+2 : \text{store } l; \\ \ulcorner e''+3, \mathcal{STMT}_2, e-2\urcorner_{\mathtt{m}}; \\ e-1 : \text{load } l; \\ e : \text{athrow} \end{array} , \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})], \mathtt{m})
\end{array}
$$

$$
\begin{array}{l}
wp^{bc}_{stmt}(\ulcorner s, \mathtt{throw}\ \mathcal{E}^{src}, e\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = \\
\quad wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{src}, e-1\urcorner_{\mathtt{m}}, wp^{bc}(e/\text{athrow}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}), \mathsf{excPost}^{src2bc}, \mathtt{m})
\end{array}
$$

$$
\begin{array}{l}
\qquad\qquad\quad \mathtt{while}\ (\mathcal{E}^{\mathcal{R}}) \\
wp^{bc}_{stmt}(\ulcorner s, \quad [\mathtt{INV}, \mathtt{modif}] \quad , e\urcorner_{\mathtt{m}}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = \\
\qquad\quad \{\mathcal{STMT}\} \\
\quad \mathtt{INV} \\
\quad \wedge \\
\quad \forall mod \in \mathtt{modif}, \\
\quad \mathtt{INV} \Rightarrow \\
\quad wp^{bc}_{seq}(\ulcorner e'+1, \mathcal{E}^{\mathcal{R}}, e-1\urcorner_{\mathtt{m}}, \\
\qquad\quad \mathcal{R}(\mathbf{st}(\mathbf{cntr}), \mathbf{st}(\mathbf{cntr}-1)) \Rightarrow \\
\qquad\qquad\quad wp^{bc}_{stmt}(\ulcorner s+1, \mathcal{STMT}, e'\urcorner_{\mathtt{m}}, \mathtt{INV}, \mathsf{excPost}^{src2bc}, \mathtt{m}) \\
\qquad\quad \wedge \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad , \\
\qquad\quad \neg\mathcal{R}(\mathbf{st}(\mathbf{cntr}), \mathbf{st}(\mathbf{cntr}-1)) \Rightarrow \psi \\
\qquad\quad \mathsf{excPost}^{src2bc}, \mathtt{m})
\end{array}
$$

Figure 7.15: DEFINITION OF wp^{bc}_{stmt}

Chapter 8

Constraint memory consumption policies using Hoare logics

8.1 Modeling memory consumption

In this chapter, we shall see how a bytecode verification condition generator can be used for checking a software component w.r.t. to bounded memory consumption policies. We propose also an annotation inference algorithm against such policies.

The remainder of the chapter is organized as follows. In section 8.2, we begin by describing the principles of our approach. Section 8.3 illustrates the approach with several examples on recursive methods, exception, inheritance. Section 8.4.1 presents an algorithm for inferring method specification for constraint memory consumption policies in terms of preconditions, postconditions and loop invariants. Finally, section 8.5 presents an overview of related works.

8.2 Principles

Let us begin with a very simple memory consumption policy which aims at enforcing that programs do not consume more than some fixed amount of memory `Max`. To enforce this policy, we first introduce a ghost variable `MemUsed` that represents at any given point of the program the memory used so far. Then, we annotate the program both with the policy and with additional statements that will be used to check that the application respects the policy.

The precondition of the method m should ensure that there must be enough free memory for the method execution. Suppose that we know an upper bound of the allocations done by method m in any execution. We will

denote this upper bound by `methodConsumption`(m). Thus there must be at least `methodConsumption`(m) free memory units from the allowed `Max` when method m starts execution. Thus the precondition for the method m is:

$$\textbf{requires } \texttt{MemUsed} + \texttt{methodConsumption}(\text{ m }) \leq \texttt{Max}.$$

The precondition of the program entry point (i.e., the method from which an application may start its execution) should state that the program has not allocated any memory, i.e. require that variable `MemUsed` is 0:

$$\textbf{requires } \texttt{MemUsed} == 0.$$

The normal postcondition of the method m must guarantee that the memory allocated during a normal execution of m is not more than some fixed number `methodConsumption`(m) of memory units. Thus for the method m the postcondition is:

$$\textbf{ensures } \texttt{MemUsed} \leq \textbf{\textbackslash old(}\texttt{MemUsed}\textbf{)} + \texttt{methodConsumption}(\text{ m }).$$

The exceptional postcondition of the method m must say that the memory allocated during an execution of m that terminates by throwing an exception `Exception` is not more than `methodConsumption`(m) units. Thus for the method m the exceptional postcondition is:

$$\textbf{exsures}\texttt{Exception } \texttt{MemUsed} \leq \textbf{\textbackslash old(}\texttt{MemUsed}\textbf{)} + \texttt{methodConsumption}(\text{ m }).$$

Loops must also be annotated with appropriate invariants. Let us assume that loop l iterates no more than $\texttt{iter}(l)$ and let $\texttt{loopConsumption}(l)$ be an upper bound of the memory allocated per iteration in l. Below we give a general form of loop specification w.r.t. the property for constraint memory consumption. The loop invariant of a loop l states that at every iteration the loop body is not going to allocate more than $\texttt{loopConsumption}(l)$ memory units and that the iterations are no more than $\texttt{iter}(l)$. We also declare an expression which guarantees loop termination, i.e. a variant (here an integer expression whose values decrease at every iteration and is always bigger or equal to 0).

$$\begin{array}{ll} \textbf{modifies} & i, \texttt{MemUsed} \\ \texttt{INV}: & \texttt{MemUsed} \leq \texttt{MemUsed}^{Before_l} + i * \texttt{loopConsumption}(l) \\ & \wedge \\ & i \leq \texttt{iter}(l) \\ \texttt{variant}: & \texttt{iter}(l) - i \end{array}$$

A special variable appears in the invariant, $\texttt{MemUsed}^{Before_l}$. It denotes the value of the consumed memory just before entering for the first time the loop l. At every iteration the consumed memory must not go beyond the upper bound given for the body of loop.

For every instruction that allocates memory the ghost variable `MemUsed` must also be updated accordingly. For the purpose of this paper, we only consider dynamic object creation with the bytecode new; arrays are left for future work and briefly discussed in the conclusion.

The function `allocInstance` : $Class \rightarrow int$ gives an estimation of the memory used by an instance of a class. Note that the memory allocated for a class instance is specific to the implementation of the virtual machine. At every program point where a bytecode new `A` is found, the ghost variable `MemUsed` must be incremented by `allocInstance`(A). This is achieved by inserting a ghost assignment immediately after any new instruction, as shown below:

```
new A
//set MemUsed = MemUsed+allocInstance(A).
```

8.3 Examples

We illustrate hereafter our approach by several examples.

8.3.1 Inheritance and overridden methods

Overriding methods are treated as follows: whenever a call is performed to a method m , we require that there is enough free memory space for the maximal consumption by all the methods that override or are overridden by m . In Fig. 8.1 we show a class `A` and its extending class `B`, where `B` overrides the method m from class `A`. Method m is invoked by n. Given that the dynamic type of the parameter passed to n is not known, we cannot know which of the two methods will be invoked. This is the reason for requiring enough memory space for the execution of any of these methods.

8.3.2 Recursive Methods

In Fig. 8.2 the bytecode of the recursive method `m` and its specification is shown. We show a simplified version of the bytecode; we assume that the constructors for the class `A` and `C` do not allocate memory. Besides the precondition and the postcondition, the specification also includes information about the termination of the method: `variant` **reg**(1), meaning that the local variable **reg**(1) decreases on every recursive call down to and no more than 0, guaranteeing that the execution of the method will terminate.

We explain first the precondition. If the condition of line `1` is not true, the execution continues at line `2`.

In the sequential execution up to line `7`, the program allocates at most `allocInstance`(A) memory units and decrements by 1 the value of **reg**(1). The instruction at line `8` is a recursive call to `m`, which either will take the same branch if **reg**(1) > 0 or will jump to line `12` otherwise, where it allocates at most `allocInstance`(A) + `allocInstance`(C) memory units. On returning from the recursive call one more allocation will be performed at line `9`. Thus

```
Specification of method
 m in class A:

requires MemUsed + k   <= Max
modifies MemUsed
ensures  MemUsed   <= \old(MemUsed) + k
```

```
Specification for method m in class B:

requires MemUsed + l   <= Max
modifies MemUsed
ensures  MemUsed   <=  \old(Mem) + l
```

```
void method n(A a)
...
//{ prove MemUsed <= Mem +max(l,k) }
invokevirtual m A
//{ assume MemUsed <= \old{MemUsed} + max(l,k)}
...
```

Figure 8.1: EXAMPLE OF OVERRIDDEN METHODS

`m` will execute, **reg**(1) times, the instructions from lines `4` to `35`, and it finally will execute all the instructions from lines `12` to `16`. The postcondition states that the method will perform no more than **\old(reg**(1)**)** recursive calls (i.e., the value of the register variable in the pre-state of the method) and that on every recursive call it allocates no more than two instances of class `A` and that it will finally allocate one instance of class `A` and another of class `C`.

8.3.3 More precise specification

We can be more precise in specifying the precondition of a method by considering what are the field values of an instance, for example. Suppose that we have the method m as shown in Fig. 8.3. We assume that in the constructor of the class `A` no allocations are done. The first line of the method m initializes one of the fields of field `b`. Since nothing guarantees that field `b` is not **null**, the execution may terminate with `NullPointerException`. Depending on the values of the parameters passed to m , the memory allocated will be different. The precondition establishes what is the expected space of free resources depending on if the field `b` is **null**or not. In particular we do not require anything for the

```
 requires  MemUsed +
           reg(1)*2*allocInstance(A) +
           allocInstance(A) +
           allocInstance(C) <=  Max

 variant   reg(1)

 ensures   reg(1) >= 0
           &&
           MemUsed <= old(MemUsed)+
           \old(reg(1))*2*allocInstance(A)+
           allocInstance(A) +
           allocInstance(C)

public void m()
//local variable loaded on
//the operand stack of method m
 0 load 1
// if \reg(1) <= 0 go to 12
 1 ifle 12}
 2 new A
// here reg(1) > 0
//@ set MemUsed =
       MemUsed +  allocInstance(A)
 3 invokespecial A.init
 4 aload 0
 5 iload 1
 6 iconst 1
// reg(1) decremented with 1
 7 isub
//recursive call with the new
//value of reg(1)
 8 invokevirtual D.m
 9 new A
//set MemUsed = MemUsed +
      allocInstance(A)
 10 invokespecial A.init
 11 goto 16
 target of the jump at  1
 12 new A
//set MemUsed = MemUsed +
      allocInstance(A)
 13 invokespecial A.init
 14 new  C
//set MemUsed =
      MemUsed +  allocInstance(C)
 15 invokespecial C.init
 16 return
```

```
public class D {
  public void m( int i) {
    if (i > 0) {
      new A();
      m(i - 1);
      new A();
    } else {
      new C();
```

free memory space in the case when `b` is **null**. In the normal postcondition we state that the method has allocated an object of class `A`. The exceptional postcondition states that no allocation is performed if `NullpointerException` causes the execution termination.

requires	$\mathbf{reg}(1) != \mathbf{null} \Rightarrow$ `MemUsed` + `allocInstance`$(A) \leq$ `Max`
modifies	`MemUsed`
ensures	`MemUsed` $\leq$ **\old(**`MemUsed`**)** + `allocInstance`(A)
exsures$NullPointerExc$	`MemUsed` $==$ **\old(**`MemUsed`**)**

```
 0 load 0
 1 getfield C.b
 2 load 2
 3 putfield B.i
 4 new A
//set MemUsed =
  MemUsed +
  allocInstance(A)
 5 dup
 6 invokespecial A.init
 7 store 1
 8 return
```

```
public class C{
 B b;
 public void m(A a,int i){
   b.i = i ;
   a = new A();
 }
}
```

Figure 8.3: Example of a method with possible exceptional termination

8.4 Inferring memory allocation for methods

In the previous section, we have described how the memory consumption of a program can be modeled in BML and verified using an appropriate verification environment. While our examples illustrate the benefits of our approach, especially regarding the precision of the analysis, the applicability of our method is hampered by the cost of providing the annotations manually. In order to reduce the burden of manually annotating the program, one can rely on annotation assistants that infer automatically some of the program annotations (indeed such assistants already exist for loop invariants, loop variants, or class invariants). In this section, we describe an implementation of an annotation assistant dedicated to the analysis of memory consumption, and illustrate its working on an example.

8.4.1 Annotation assistant

The inference algorithm proposed here works on programs without recursive methods, exception throwing and handling.

The user must provide information about the memory required to create objects of the given classes, i.e. he must give the definition of the function `allocInstance`$(\cdot)$. The variant and the maximum number of iterations `iter`(l) for every loop l are either given by the user or are synthesized through appropriate mechanisms.

Based on this information, the annotation assistant inserts the ghost assignments on appropriate places, and then computes recursively the memory allocated on each loop and method. A pseudo-code of the algorithm for inferring an upper bound for method allocations is given in Fig. 8.4. Essentially, it finds the maximal memory that can be allocated in a method by exploring all its possible execution paths. The algorithm for exploring an execution path starts from the last instruction in the path (i.e. a return instruction). The algorithm use standars techniques to detect the loop entry instructions. For each loop entry instruction it also finds the set of the corresponding "loop end" instructions i.e. the instructions that target to and which are dominated by the loop entry instruction; you are not entering in detail in the loop detection algorithm as it is standard and the reader may see Section 10 of [7] for a description of the algorithms.

function `methodConsumption`(.)
Input: Bytecode of a method `m` .
Output: Upper bound of the memory allocated by `m` .
Body:

1. Detect all the loops in `m`; for every loop l determine `loopSet`(l), `entry`(l) and `loopEndSet`(l);
2. Apply the function `allocPath` to each instruction i_k, such that $i_k =$ return;
3. Take the maximum of the results given in the previous step: $max_{i_k=\text{return}}$`allocPath`(i_k).

Figure 8.4: Inference algorithm

The auxiliary function *allocPath*, which infers the maximal allocations done by the set of execution paths ending with the same return instruction, is given in Fig. 8.5. Inferring the memory allocated inside loops is done by the function `allocLoopPath`$(\cdot,\cdot)$, which is invoked by `allocPath` whenever the current instruction belong to a loop. The specification of the function is shown in Fig. 8.6.

The annotation assistant currently synthesize only simple memory policies

$$\texttt{allocPath}(i_s) =$$

$$\begin{cases} alloc\_instr(i_s) & if\ i_s\ has\ no\ predecessors \\ \texttt{loopConsumption}(\texttt{entry}(l)) + max_{i_k \in \texttt{preds}(i_s) - \texttt{loopEndSet}(l)}(\texttt{allocPath}(i_k)) & if\ i_s \in \texttt{loopSet}(l) \\ alloc\_instr(i_s) + max_{i_k \in \texttt{preds}(i_s)}(\texttt{allocPath}(i_k)) & else \end{cases}$$

Figure 8.5: DEFINITION OF THE FUNCTION $\texttt{allocPath}(i_s)$

$$\texttt{loopConsumption}(\texttt{entry}(l)) =$$

$$\texttt{iter}(l) * max_{i_e \in \texttt{loopEndSet}(l)}(\texttt{allocLoopPath}(\texttt{entry}(l), i_e))$$

$$\texttt{allocLoopPath}(\texttt{entry}(l), i_s) =$$

$$\begin{cases} alloc\_instr(\texttt{entry}(l)) & if\ i_s = \texttt{entry}(l) \\ \texttt{loopConsumption}(\texttt{entry}(l')) + max_{i_k \in \texttt{preds(entry}(l')) - \texttt{loopEndSet}(l')}(\texttt{allocLoopPath}(\texttt{entry}(l), i_k)) & if\ i_s \in \texttt{loopSet}(l')\ l'\ is\ nested\ in\ l \\ alloc\_instr(i_s) + max_{i_k \in \texttt{preds}(i_s)}(\texttt{allocLoopPath}(\texttt{entry}(l), i_k)) & else \end{cases}$$

Figure 8.6: DEFINITION OF THE FUNCTION $\texttt{loopConsumption}(\cdot)$ AND $\texttt{allocLoopPath}(\cdot, \cdot)$

(i.e., whenever the memory consumption policy does not depend on the values of inputs). Furthermore, it does not deal with arrays, subroutines, nor exceptions. Our approach may be extended to treat such cases For sake of simplicity, we have also restricted the loop analysis only to those with a unique entry point, which is the case for code produced by non-optimizing compilers. Dealing with loops with a multiple entry points is left for a future work.

8.5 Related work

must look at the MRG project in more detail

The use of type systems has been a useful tool for guaranteeing that well typed programs run within stated space-bounds. Previous work along these lines defined typed assembly languages, inspired on [52] while others emphasised the use of type systems for functional languages [5, 36, 38].

For instance in [4] the authors present a first-order linearly typed assembly language which allows the safe reuse of heap space for elements of different types. The idea is to design a family of assembly languages which have high-level typing features (e.g. the use of a special *diamond* resource type) which are used to express resource bound constraints. Closely related to the previous-mentioned paper, [70] describes a type theory for certified code, in which type safety guarantees cooperation with a mechanism to limit the CPU usage of untrusted code. Another recent work is [3] where the resource bounds problem is studied in a simple stack machine. The authors show how to perform type, size and termination verifications at the level of the byte-code.

An automatic heap space usage static analysis for first-order functional programs is given in [37]. The analysis both determines the amount of free cells necessary before execution as well as a safe (under)-estimate of the size of a *free-list* after successful execution of a function. These numbers are obtained as solutions to a set of linear programming (LP) constraints derived from the program text. Automatic inference is obtained by using standard polynomial-time algorithms for solving LP constraints. The correctness of the analysis is proved with respect to an operational semantics that explicitly keeps track of the memory structure and the number of free cells.

A logic for reasoning about resource consumption certificates of higher-order functions is defined in [24]. The certificate of a function provides an over-approximation of the execution time of a call to the function. The logic only defines what is a correct deduction of a certificate and has no inference algorithm associated with it. Although the logic is about computation time the authors claim it could be extended to measure memory consumption.

Another mechanical verification of a byte code language is [21], where a constraint-based algorithm is presented to check the existence of new instructions inside intra- and inter-procedural loops. It is completely formalised in Coq and a certified analyser is obtained using Coq's extraction mechanism. The time complexity of such analysis performs quite good but the auxiliary memory used does not allow it to be on-card. Their analysis is less precise than ours, since they work on an abstraction of the execution traces not considering the number

of times a cycle is iterated (there are no annotations). Along these lines, a similar approach has been followed by [66]; no mechanical proof nor implementation is provided in such work.

Other related research direction concerns runtime memory analysis. The work [31] presents a method for analysing, monitoring and controlling dynamic memory allocation, using pointer and scope analysis. By instrumenting the source code they control memory allocation at run-time. In order to guarantee the desired memory allocation property, in [29] is implemented a runtime monitor to control the execution of a Java Card applet. The applet code is instrumented: a call to a monitor method is added before a new instruction. Such monitor method has as parameter the size of the allocation request and it halts the execution of the applet if a predefined allocation bound is exceeded.

Upon completion of this work we became aware of a recent, and still unpublished, result along the same lines of ours. Indeed, a hybrid (i.e., static and dynamic) resource bound checker for an imperative language designed to admit decidable verification is presented in [22]. The verifier is based on a variant of Dijkstra's weakest precondition calculus using "generalized predicates", which keeps track of the resource units available. Besides adding loop invariants, pre- and post-conditions, the programmer must insert "acquires" annotations to reserve the resource units to be consumed. Our approach has the advantage of treating recursive methods and exceptions, not taken into account in [22]. Another difference with our work is that we operate on the bytecode instead of on the source code.

Chapter 9

A Low-Footprint Java-to-Native Compilation Scheme Using Formal Methods

In this chapter, we will focus on the use of our verification scheme in Java native compiler optimizations. Let us first see what is the context and motivations for applying formal program verification in compiler optimization.

Enabling Java on embedded and restrained systems is an important challenge for today's industry and research groups [53]. Java brings features like execution safety and low-footprint program code that make this technology appealing for embedded devices which have obvious memory restrictions, as the success of Java Card witnesses. However, the memory footprint and safety features of Java come at the price of a slower program execution, which can be a problem when the host device already has a limited processing power. As of today, the interest of Java for smart cards is still growing, with next generation operating systems for smart cards that are closer to standard Java systems [43, 33], but runtime performance in still an issue. To improve the runtime performances of Java systems, a common practice is to translate some parts of the program bytecode into native code.

Doing so removes the interpretation layer and improves the execution speed, but also greatly increases the memory footprint of the program: it is expected that native code is about three to four times the size of its Java counterpart, depending on the target architecture. This is explained by the less-compact form of native instructions, but also by the fact that many safety-checks that are implemented by the virtual machine must be reproduced in the native code. For instance, before dereferencing a pointer, the virtual machine checks whether it is `null` and, if it is, throws a `NullPntrExc`. Every time a bytecode that

implements such safety-behaviors is compiled into native code, these behaviors must be reproduced as well, leading to an explosion of the code size. Indeed, a large part of the Java bytecode implement these safety mechanisms.

Although the runtime checks are necessary to the safety of the Java virtual machine, they are most of the time used as a protection mechanism against programming errors or malicious code: A runtime exception should be the result of an exceptional, unexpected program behavior and is rarely thrown when executing sane code - doing so is considered poor programming practice. The safety checks are therefore without effect most of the time, and, in the case of native code, uselessly enlarge the code size.

Several studies proposed to factorize these checks or in some case to eliminate them, but none proposed a complete elimination without hazarding the system security. In the following, we use formal proofs to ensure that run-time checks can never be true in a program, which allows us to completely and safely eliminate them from the generated native code. The programs to optimize are JML-annotated against runtime exceptions and verified by the JACK. We have been able to remove almost all of the runtime checks on tested programs, and obtained native ARM thumb code which size was comparable to the original bytecode.

The remainder of this paper is organized as follows. In section 9.1, we overview the methods used for compiling Java bytecode into native code, and evaluate the previous work aiming at optimizing runtime exceptions in the native code. Section 9.2 is a brief presentation of the runtime exceptions in Java. Then, section 9.3 describes our method for removing runtime exceptions on the basis of formal proofs. We experimentally evaluate this method in section 9.4, discuss its limitations in 9.5 and conclude in **??**.

9.1 Ahead-of-time & just-in-time compilation

Compiling Java into native code common on embedded devices. This section gives an overview of the different compilation techniques of Java programs, and points out the issue of runtime exceptions.

Ahead-of-Time (AOT) compilation is a common way to improve the efficiency of Java programs. It is related to Just-in-Time (JIT) compilation by the fact that both processes take Java bytecode as input and produce native code that the architecture running the virtual machine can directly execute. AOT and JIT compilation differ by the time at which the compilation occurs. JIT compilation is done, as its name states, just-in-time by the virtual machine, and must therefore be performed within a short period of time which leaves little room for optimizations. The output of JIT compilation is machine-language. On the contrary, AOT compilation compiles the Java bytecode way before the program is run, and links the native code with the virtual machine. In other words, it translates non-native methods into native methods (usually C code) prior to the whole system execution. AOT compilers either compile the Java program entirely, resulting in a 100% native program without a Java interpreter,

or can just compile a few important methods. In the latter case, the native code is usually linked with the virtual machine. AOT compilation have no or few time constraints, and can generate optimized code. Moreover, the generated code can take advantage of the C compiler's own optimizations.

JIT compilation in interesting by several points. For instance, there is no prior choice about which methods must be compiled: the virtual machine compiles a method when it appears that doing so is beneficial, e.g. because the method is called often. However, JIT compilation requires embedding a compiler within the virtual machine, which needs resources to work and writable memory to store the compiled methods. Moreover, the compiled methods are present twice in memory: once in bytecode form, and another time in compiled form. While this scheme is efficient for decently-powerful embedded devices such as PDAs, it is inapplicable to very restrained devices like smartcards or sensors. For them, ahead-of-time compilation is usually preferred because it does not require a particular support from the embedded virtual machine outside of the ability to run native methods, and avoids method duplication. AOT compilation has some constraints, too: the compiled methods must be known in advance, and dynamically-loading new native methods is forbidden, or at least very unsafe.

Both JIT and AOT compilers must produce code that exactly mimics the behavior of the Java virtual machine. In particular, the safety checks performed on some bytecode must also be performed in the generated code.

9.2 Java runtime exceptions

The JVM [48] specifies a safe execution environment for Java programs. Contrary to native execution, which does not automatically control the safety of the program's operations, the Java virtual machine ensures that every instruction operates safely. The Java environment may throw predefined runtime exceptions at runtime, like the following ones:

`NullPointerException` This exception is thrown when the program tries to dereference a `null` pointer. Among the instructions that may throw this exceptions are: `getfield`, `putfield`, `invokevirtual`, `invokespecial`, the set of *`type`*`astore` instructions[1] may throw such an exception.

`ArrayIndexOutOfBoundsException` If an array is accessed out of its bounds, this exception is thrown to prevent the program from accessing an illegal memory location. According to the Java Virtual Machine specification, the instructions of the family *`type`*`astore` and *`type`*`aload` may throw such an exception.

`ArithmeticException` This exception is thrown when exceptional arithmetic conditions are met. Actually, there is only one such case that may occur

[1] the JVM instructions are parametrized, thus we denote by *`type`*`astore` the set of array store instructions, which includes `iastore`, `sastore`, `lastore`, ...

during runtime, namely the division of an integer by zero, which may be done by `idiv`, `irem`, `ldiv` and `lrem`.

`NegativeArraySizeException` Thrown when trying to allocate an array of negative size. `newarray`, `anewarray` and `multianewarray` may throw such an exception.

`ArrayStoreException` Thrown when an object is attempted to be stored into an array of incompatible type. This exception may be thrown by the `aastore` instruction.

`ClassCastException` Thrown when attempting to cast an object to an incompatible type. The `checkcast` instruction may throw such an exception.

`IllegalMonitorStateException` Thrown when the current thread is not the owner of a released monitor, typically by `monitorexit`.

If the JVM detects that executing the next instruction will result in an inconsistency or an illegal memory access, it throws a runtime exception, that may be caught by the current method or by other methods on the current stack. If the exception is not caught, the virtual machine exits. This safe execution mode implies that many checks are made during runtime to detect potential inconsistencies. For instance, the `aastore` bytecode, which stores an object reference into an array, may throw three different exceptions:

- `NullPointerException`, if the reference to the array is `null`,
- `ArrayIndexOutOfBoundsException`, if the index in which to store the object is not within the bounds of the array,
- `ArrayStoreException`, if the object to store is not assignment-compatible with the array (for instance, storing an `Integer` into an array of `Boolean`).

Of the 202 bytecodes defined by the Java virtual machine specification, we noticed that 43 require at least one runtime exception check before being executed. While these checks are implicitly performed by the bytecode interpreter in the case of interpreted code, they must explicitly be issued every time such a bytecode is compiled into native code, which leads to a code size explosion. Ishizaki et al. measured that bytecodes requiring runtime checks are frequent in Java programs: for instance, the natively-compiled version of the SPECjvm98 `compress` benchmark has 2964 exception check sites for a size of 23598 bytes. As for the `mpegaudio` benchmark, it weights 38204 bytes and includes 6838 exception sites [40]. The exception check sites therefore make a non-neglectable part of the compiled code.

Figure 9.1 shows an example of Java bytecode that requires a runtime check to be issued when being compiled into native code.

It is, however, possible to eliminate these checks from the native code if the execution context of the bytecode shows that the exceptional case never happens. In the program of figure 9.1, the lines 2 and 3 could have been omitted

Java version:

```
iload i
iload j
idiv
ireturn
```

C version:

```
1 int i, j;
2 if (j == 0)
3   THROW(ArithmeticException);
4 RETURN_INT(i / j);
```

Figure 9.1: A Java bytecode program and its (simplified) C-compiled version. The behavior of the division operator in Java must be entirely reproduced by the C program, which leads to the generation of a runtime exception check site

if we were sure that for all possible program paths, `j` can never be equal to zero at this point. This allows to generate less code and thus to save memory. Removing exception check sites is a topic that has largely been studied in the domain of JIT and AOT compilation.

9.3 Optimizing ahead-of-time compiled Java code

For verifying the bytecode that will be compiled into native code, we use the JACK verification framework. In particular, we use the compiler from JML to BML and the bytecode verification condition generator.

Verifying that a bytecode program does not throw Runtime exceptions using JACK involves several stages:

1. writing the JML specification at the source level of the application, which expresses that no runtime exceptions are thrown.

2. compiling the Java sources into bytecode

3. compiler the JML specification into BML specification and add it in the class file

4. generating the verification conditions over the bytecode and its BML specification, and proving the verification conditions 9.3.2. During the calculation process of the verification conditions, they are indexed with the index of the instruction in the bytecode array they refer to and the type of specification they prove (e.g. that the proof obligation refers to the exceptional postcondition in case an exception of type `Exc` is thrown when executing the instruction at index `i` in the array of bytecode instructions of a given method). Once the verifications are proved, information about which instructions can be compiled without runtime checks is inserted in user defined attributes of the class file.

5. using these class file attributes in order to optimize the generated native code. When a bytecode that has one or more runtime checks in its semantics is being compiled, the bytecode attribute is checked in order to make sure that the checks are necessary. It indicates that the exceptional condition has been proved to never happen, then the runtime check is not generated.

Our approach benefits from the accurateness of the JML specification and from the bytecode verification condition generator. Performing the verification over the bytecode allows to easily establish a relationship between the proof obligations generated over the bytecode and the bytecode instructions to optimized.

In the rest of this section, we explain in detail all the stages of the optimization procedure.

9.3.1 Methodology for writing specification against runtime exception

We now illustrate with an example what are the JML annotations needed for verifying that a method does not throw a runtime exception. Figure 9.2[2] shows a Java method annotated with a JML specification. The method `clear` declared in class `Code_Table` receives an integer parameter `size` and assigns `0` to all the elements in the array field `tab` whose indexes are smaller than the value of the parameter `size`. The specification of the method guarantees that if every caller respects the method precondition and if every execution of the method guarantees its postcondition then the method `clear` never throws an exception of type or subtype `java.lang.Exception`[3]. This is expressed by the class and method specification contracts. First, a class invariant is declared which states that once an instance of type `Code_Table` is created, its array field `tab` is not null. The class invariant guarantees that no method will throw a `NullPntrExc` when dereferencing (directly or indirectly) `tab`.

The method precondition requires the `size` parameter to be smaller than the length of `tab`. The normal postcondition, introduced by the keyword `ensures`, basically says that the method will always terminate normally, by declaring that the set of final states in case of normal termination includes all the possible final states, i.e. that the predicate `true` holds after the method's normal execution[4]. On the other hand, the exceptional postcondition for the exception `java.lang.Exception` says that the method will not throw any exception of type `java.lang.Exception` (which includes all runtime exceptions). This is done by declaring that the set of final states in the exceptional termination case is empty, i.e. the predicate `false` holds if an exception caused the termination

[2] although the analysis that we describe is on bytecode level, for the sake of readability, the examples are also given on source level

[3] Note that every Java runtime exception is a subclass of `java.lang.Exception`

[4] Actually, after terminating execution the method guarantees that the first `size` elements of the array tab will be equal to 0, but as this information is not relevant to proving that the method will not throw runtime exceptions we omit it

```
final class Code_Table {
  private/*@spec_public */short tab[];

  //@invariant tab != null;

  ...

  //@requires size <= tab.length;
  //@ensures true;
  //@exsures (Exception) false;
  public void clear(int size) {
  1  int code;
  2  //@loop_modifies code, tab[*];
  3  //@loop_invariant code <= size && code >= 0;
  4  for (code = 0; code < size; code++) {
  5    tab[code] = 0;
     }
  }
}
```

Figure 9.2: A JML-annotated method

of the method. The loop invariant says that the array accesses are between index `0` and index `size - 1` of the array `tab`, which guarantees that no loop iteration will cause a `ArrayIndexOutOfBoundsException` since the precondition requires that `size <= tab.length`.

9.3.2 From program proofs to program optimizations

In this phase, the bytecode instructions that can safely be executed without runtime checks are identified. Depending on the complexity of the verification conditions, Jack can discharge them to the fully automatic prover Simplify, or to the Coq and AtelierB interactive theorem prover assistants.

There are several conditions to be met for a bytecode instruction to be optimized safely – the precondition of the method the instruction belongs to must hold every time the method is invoked, and the verification condition related to the exceptional termination must also hold. In order to give a flavor of the verification conditions we deal with, figure 9.3 shows part of the verification condition related to the possible `ArrayIndexOutOfBounds` exceptional termination of instruction `11 sastore` in figure 3, which is actually provable.

Once identified, proved instructions can be marked in user-defined attributes of the class file so that the compiler can find them.

$$
\begin{array}{l}
\ldots \\
\texttt{length}(\texttt{tab}(\mathbf{reg}(0)) \leq \mathbf{reg}(2)_{15} \vee \mathbf{reg}(2)_{15} < 0 \\
\wedge \\
\mathbf{reg}(2)_{15} \geq 0 \\
\wedge \\
\mathbf{reg}(2)_{15} < \mathbf{reg}(1) \\
\wedge \\
\mathbf{reg}(1) \leq \texttt{length}(\texttt{tab}(\mathbf{reg}(0)))
\end{array}
\Rightarrow \mathit{false}
$$

Figure 9.3: THE VERIFICATION CONDITION FOR THE ARRAYINDEXOUTOFBOUNDEXCEPTION CHECK RELATED TO THE SASTORE INSTRUCTION OF FIGURE 3

9.4 Experimental results

This section presents an application and evaluation of our method on various Java programs.

9.4.1 Methodology

We have measured the efficiency of our method on two kinds of programs, that implement features commonly met in restrained and embedded devices. **crypt** and **banking** are two smartcard-range applications. **crypt** is a cryptography benchmark from the Java Grande benchmarks suite, and **banking** is a little banking application with full JML annotations used in [20]. **scheduler** and **tcpip** are two embeddable system components written in Java, which are actually used in the JITS [1] platform. **scheduler** implements a threads scheduling mechanism, where scheduling policies are Java classes. **tcpip** is a TCP/IP stack entirely written in Java, that implements the TCP, UDP, IP, SLIP and ICMP protocols. These two components are written with low-footprint in mind ; however, the overall system performance would greatly benefit from having them available in native form, provided the memory footprint cost is not too important.

For every program, we have followed the methodology described in section 9.3 in order to prove that runtime exceptions are not thrown in these programs. We look at both the number of runtime exception check sites that we are able to remove from the native code, and the impact on the memory footprint of the natively-compiled methods with respect to the unoptimized native version and the original bytecode. The memory footprint measurements were obtained by compiling the C source file generated by the JITS AOT compiler using GCC 4.0.0 with optimization option `-Os`, for the ARM platform in thumb mode. The native methods sizes are obtained by inspecting the .o file with `nm`, and getting the size for the symbol corresponding to the native method.

Table 9.1: Number of exception check sites and memory footprints when compiled for ARM thumb

Program	# of exception check sites			Memory footprint (bytes)		
	Bytecode	JC	Proven AOT	Bytecode	Naive AOT	Proven AOT
crypt	190	79	1	1256	5330	1592
banking	170	12	0	2320	5634	3582
scheduler	215	25	0	2208	5416	2504
tcpip	1893	288	0	15497	41540	18064

Regarding the number of eliminated exception check sites, we also compare our results with the ones obtained using the JC virtual machine mentioned in 9.6, version 1.4.6. The results were obtained by running the `jcgen` program on the benchmark classes, and counting the number of explicit exception check sites in the generated C code. We are not comparing the memory footprints obtained with the JITS and JC AOT compilers, for this result would not be pertinent. Indeed, JC and JITS have very different ways to generate native code. JITS targets low memory footprint, and JC runtime performance. As a consequence, a runtime exception check site in JC is heavier than one in JITS, which would falsify the experiments. Suffices to say that our approach could be applied on any AOT compiler, and that the most relevant measurement is the number of runtime exception check sites that remains in the final binary - our measurements on the native code memory footprint are just here to evaluate the size impact of exception check sites.

9.4.2 Results

Table 9.1 shows the results obtained on the four tested programs. The three first columns indicate the number of check sites present in the bytecode, the number of explicit check sites emitted by JC, and the number of check sites that we were unable to prove useless and that must be present in our optimized AOT code. The last columns give the memory footprints of the bytecode, unoptimized native code, and native code from which all proved exception check sites are removed.

On all the tested programs, we were able to prove that all but one exception check site could be removed. The only site that we were unable to prove from **crypt** is linked to a division, which divisor is a computed value that we were unable to prove not equal to zero. JC has to retain 16% of all the exception check sites, with a particular mention for **crypt**, which is mainly made of array accessed and had more remaining check sites.

The memory footprints obtained clearly show the heavy overhead induced by exception check sites. Despite of the fact that the exception throwing convention has deliberately been simplified for our experiments, optimized native code is

Table 9.2: Human work on the tested programs

Program	Source code size (bytes)		Proved lemmas	
	Code	JML	Automatically	Manually
crypt	4113	1882	227	77
banking	11845	15775	379	159
scheduler	12539	3399	226	49
tcpip	83017	15379	2233	2191

less than half the size of the non-optimized native code. The native code of **crypt**, which heavily uses arrays, is actually made of exception checking code at 70%.

Comparing the size of the optimized native versions with the bytecode reveals that proved native code is just slightly bigger than bytecode. The native code of **crypt** is 27% bigger than its bytecode version. Native **scheduler** only weights 13.5% more that its bytecode, **tcpip** 16.5%, while **banking** is 54% heavier. This last result is explained by the fact that, being an application and not a system componant, **banking** includes many native-to-java method invokations for calling system services. The native-to-java calling convention is costly in JITS, which artificially increases the result.

Finally, table 9.2 details the human work required to obtain the proofs on the benchmark programs, by comparing the amount of JML code with respect to the comments-free source code of the programs. It also details how many lemmas had to be manually proved.

On the three programs that are annotated for the unique purpose of our study, the JML overhead is about 30% of the code size. The **banking** program was annotated in order to prove other properties, and because of this is made of more JML annotations than actual code. Most of the lemmas could be proved by Simplify, but a non-neglectable part needed human-assistance with Coq. The most demanding application was the TCP/IP stack. Because of its complexity, nearly half of the lemmas could not be proved automatically.

The gain in terms of memory footprint obtained using our approach is therefore real. One may also wonder whether the runtime performance of such optimized methods would be increased. We did the measurements, and only noticed a very slight, almost undetectable, improvement of the execution speed of the programs. This is explained by the fact that the exception check sites conditions are always false when evaluated, and therefore the amount of supplementary code executed is very low. The bodies of the proved runtime exception check sites are, actually, dead code that is never executed.

9.5 Limitations

Our approach suffers from some limitations and usage restrictions, regarding its application on multi-threaded programs and in combination with dynamic code loading.

9.5.1 Multi-threaded programs

As we said in section 9.3, JACK only supports the sequential subset of Java. Because of this, we are unable to prove check sites related to monitor state checking, that typically throws an `IllegalMonitorStateException`. However, they can be simplified if it is known that the system will never run more than one thread simultaneously. It should be noted, that Java Card does not make use of multi-threading and thus doesn't suffer from this limitation.

9.5.2 Dynamic code loading

Our removal of runtime exception check sites is based on the assumption that a method's preconditions are always respected at all its call sites. For closed systems, it is easy to verify this property, but in the case of open systems which may load and execute any kind of code, the property could not always be ensured. In the case where the set of applications that will run on the system is not statically known, our approach could not be safely applied on public methods since dynamically-loaded code may call them without respecting their preconditions. However, a solution is to verify the methods of every dynamically loaded class before it is loaded w.r.t. the specification of the classes already installed classes and their methods.

9.6 Related work

Toba [58] is a Java-to-C compiler that transforms a whole Java program into a native one. Harissa [54] is a Java environment that includes a Java-to-C compiler as well as a virtual machine, and therefore supports mixed execution. While both environments implement some optimizations, they are not able to detect and remove unused runtime checks during ahead-of-time compilation. The JC Virtual Machine [2] is a Java virtual machine implementations that converts class files into C code using the Soot [69] framework, and runs their compiled version. It supports redundant exceptions checks removal, and is tuned for runtime performance, by using operating system signals in order to detect exceptional conditions like null pointer dereferencing. This allows to automatically remove most of the `NullPointerException`-related checks.

In [39] and [8], Hummel et al. use a Java compiler that annotates bytecodes with higher-level information known during compile-time in order to improve the efficiency of generated native code. [40] proposes methods for optimizing exceptions handling in the case of JIT compiled native code. These works rely

on knowledge that can be statically inferred either by the Java compiler or by the JIT compiler. In doing so, they manage to efficiently factorize runtime checks, or in some cases to remove them. However, they are still limited to the context of the compiled method, and do not take the whole program into account. Indeed, knowing properties about a the parameters of a method can help removing further checks.

We propose to go further than these approaches, by giving more precise directives as to how the program behaves in the form of JML annotations. These annotations are then used to get formal behavioral proofs of the program, which guarantee that runtime checks can safely be eliminated for ahead-of-time compilation.

Chapter 10

Conclusion

.1 Proofs of properties from Section 7.3.4

Property .1.0.1 (Property 7.3.4.2 Compilation of statements and expressions) *For any statement or expression $\mathcal{S}$, start label s and end label e, the compiler will produce a list of bytecode instruction $\ulcorner s, \mathcal{S}, e \urcorner_{\mathtt{m}}$ such that:*

$$
\begin{array}{l}
\forall i, (i \in \ulcorner s, \mathcal{S}, e \urcorner_{\mathtt{m}}) \wedge \\
\quad (i \rightarrow k) \wedge \\
\quad \neg (k \in \ulcorner s, \mathcal{S}, e \urcorner_{\mathtt{m}}) \\
\quad \neg \mathit{isExcHandlerStart}(k) \Rightarrow \\
\qquad k = e + 1
\end{array}
$$

The proof is done by induction on the structure of the compiled statement. We scatch the proof for the compilation of the if statement, the rest of the cases being similar

Proof:

{ *Assume that* $\exists i, i \in [s \ldots e], \exists k, k \notin [s \ldots e+1] \wedge i \rightarrow k$ }

{ *by definition of the compiler function for if statements in section* **??** }
$\ulcorner s, \texttt{if}\ (\mathcal{E}^{\mathcal{R}})\ \texttt{then}\ \{\mathcal{STMT}_1\}\ \texttt{else}\ \{\mathcal{STMT}_2\}, e \urcorner_{\texttt{m}} =$
$\ulcorner s, \mathcal{E}^{\mathcal{R}}, e' \urcorner_{\texttt{m}}$;
$e' + 1$ if_cond $e'' + 2$;
$\ulcorner e' + 2, \mathcal{STMT}_2, e'' \urcorner_{\texttt{m}}$
$e'' + 1$ goto $e + 1$;
$\ulcorner e'' + 2, \mathcal{STMT}_1, e \urcorner_{\texttt{m}}$;

(1) { *Assume that* $s \leq i \leq e'$ }
{ *by induction hypothesis for* $\mathcal{E}^{\mathcal{R}}$ *we get* }
(2)$\forall i, s \leq i \leq e',\ i \rightarrow k) \wedge$
$\neg(k \in \ulcorner s, \mathcal{STMT}, e \urcorner_{\texttt{m}})$
$\neg isExcHandlerStart(k) \Rightarrow$
$k = e' + 1$
{ *From (1) and (2) we get a contradiction in this case*}

(3) { *Assume that* $e' + 2 \leq i \leq e''$ }
(4) $\forall i, e' + 2 \geq i \leq e'',\ i \rightarrow k) \wedge$
$\neg(k \in \ulcorner s, \mathcal{STMT}, e \urcorner_{\texttt{m}})$
$\neg isExcHandlerStart(k) \Rightarrow$
$k = e'' + 1$
{ *From (3) and (4) we get a contradiction in this case*}

(5) { *Assume that* $e'' + 2 \leq i \leq e$ }
(6) $\forall i, e'' + 2 \geq i \leq e,\ i \rightarrow k) \wedge$
$\neg(k \in \ulcorner s, \mathcal{STMT}, e \urcorner_{\texttt{m}})$
$\neg isExcHandlerStart(k) \Rightarrow$
$k = e + 1$

{ *From (5) and (6) we get a contradiction in this case*}

(7){ *Assume that* $s = e' + 1$ }
{ *as* $e' + 1$ if_cond $e'' + 2$ *and* $e'' + 2$ *and* $e' + 2$ *are labels in the compilation of the statement, we get a contradiction in this case* }

(8){ *Assume that* $s = e'' + 1$ }
{ *as* $e'' + 1$ goto $e + 1$ *we get a contradiction once again*

Qed.

Property .1.0.2 (Compilation of expressions) *For any expression* $\mathcal{E}^{src}$*, starting label* s *and end label* e*, the compilation* $\ulcorner s, \mathcal{E}^{src}, e \urcorner_{\texttt{m}}$ *is a block of bytecode*

instruction in the sense of Def. 7.3.4.1

Proof:

Following the Def. 7.3.4.1 of block of bytecode instructions, we have to see if the compilation of an expression respects three conditions. The first condition of Def.7.3.4.1 follows from lemma 7.3.4.3. Def. 7.3.4.1 also requires that there are no jump instructions in the list of instructions representing the compilation of an expression. This is established by induction over the structure of the expression. The third condition in Def. 7.3.4.1 states that the compilation $\ulcorner s, \mathcal{E}^{src}, e \urcorner_{\mathtt{m}}$ is such that no instruction except s may be a loop entry in the sense of Def. 2.9.1 in Chapter 5, Section **??**. This is the case, as there are no jump instructions inside the compiled expression and all the instructions inside an expression are sequential.

this is not well explained

Qed.

Property .1.0.3 (Exception handler property for statements) *For every statement $\mathcal{STMT}$ which is not a try catch statement in method* m *and its strict substatement $\mathcal{STMT}'$, i.e. $\mathcal{STMT}[[\mathcal{STMT}']]$ if their respective compilations are $\ulcorner s, \mathcal{STMT}, e \urcorner_{\mathtt{m}}$ and $\ulcorner s', \mathcal{STMT}', e' \urcorner_{\mathtt{m}}$ then the following holds:*

$$\forall \mathtt{Exc}, \mathit{findExcHandler}(\mathtt{Exc}, e, \mathtt{m.excHndlS}) = \mathit{findExcHandler}(\mathtt{Exc}, e', \mathtt{m.excHndlS})$$

Proof: The proof is by contradiction. Assume this is not true, i.e.

$$\begin{array}{ll} \exists \mathcal{STMT}, \mathcal{STMT}', & \\ \textit{(1)} & \mathcal{STMT} \neq \mathtt{try}\{\ldots\}\mathtt{catch}\{\ldots\} \wedge \\ \textit{(2)} & \mathcal{STMT}[[\mathcal{STMT}']] \\ \textit{(3)} & \ulcorner s, \mathcal{STMT}, e \urcorner_{\mathtt{m}} \wedge \\ \textit{(4)} & \ulcorner s', \mathcal{STMT}', e' \urcorner_{\mathtt{m}} \wedge \\ \textit{(5)} & \exists \mathtt{Exc}, \mathit{findExcHandler}(\mathtt{Exc}, e, \mathtt{m.excHndlS}) \neq \\ & \mathit{findExcHandler}(\mathtt{Exc}, e', \mathtt{m.excHndlS}) \end{array}$$

This means that there exists two elements $(s_1, e_1, eH_1, \mathtt{Exc})$ and $(s_2, e_2, eH_2, \mathtt{Exc})$ in the exception handler table of method m m.excHndlS such that :

$$\textit{(6)} \quad eH_1 \neq eH_2$$

From lemma 7.3.4.9 we get that :

$$\begin{array}{ll} & \exists \mathcal{STMT}_1, s_1, e_1, \ulcorner s_1, \mathcal{STMT}_1, e_1 \urcorner_{\mathtt{m}} \\ \textit{(7)} & \wedge \\ & \exists \mathcal{STMT}_2, s_2, e_2, \ulcorner s_2, \mathcal{STMT}_2, e_2 \urcorner_{\mathtt{m}} \end{array}$$

From the initial condition *(2)* by lemma 7.3.4.4 we conclude that

$$\textit{(8)} \quad \ulcorner s_1, \mathcal{STMT}_1, e_1 \urcorner_{\mathtt{m}} \notin \ulcorner s, \mathcal{STMT}, e \urcorner_{\mathtt{m}}$$

Because $e \in [s_2 \ldots e_2] \wedge e' \in [s_1 \ldots e_1] \wedge e, e' \in [s \ldots e]$, *(8)* by applying Lemma 7.3.4.5 we can conclude that

$$\begin{array}{l}\ulcorner s, \mathcal{STMT}, e\urcorner_{\mathtt{m}} \in \ulcorner s_1, \mathcal{STMT}_1, e_1\urcorner_{\mathtt{m}} \in \ulcorner s_2, \mathcal{STMT}_2, e_2\urcorner_{\mathtt{m}} \\ \vee \\ \ulcorner s, \mathcal{STMT}, e\urcorner_{\mathtt{m}} \in \ulcorner s_2, \mathcal{STMT}_2, e_2\urcorner_{\mathtt{m}} \in \ulcorner s_1, \mathcal{STMT}_1, e_1\urcorner_{\mathtt{m}}\end{array}$$

In both of the cases and of the definition of *findExcHandler* in Chapter 2.8, Section 2.6 this means that

$$\mathit{findExcHandler}(\mathtt{Exc}, e, \mathtt{m}.\mathsf{excHndlS}) = \mathit{findExcHandler}(\mathtt{Exc}, e', \mathtt{m}.\mathsf{excHndlS})$$

which is in contradiction with *(6). Qed.*

.2 Proofs from Section 7.6

Lemma .2.1 (7.6.2 Wp of a compiled expression) *For any expression $\mathcal{E}^{src}$ from our source language, for any formula $\psi : \mathcal{F}$ of the source assertion language and any formula $\phi : P$ such that ϕ may only contain stack expressions of the form $\mathbf{st}(\mathbf{cntr} - k), k \geq 0$, the following holds*

There exist $Q, R : \mathcal{F}$ such that

$$\begin{array}{l} wp^{src}(\mathcal{E}^{src}, \psi, \mathsf{excPostRTE}, \mathtt{m})_v \ = \\ Q \Rightarrow \psi \\ \wedge \\ R \\ \\ \Longrightarrow \\ \\ wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{src}, e\urcorner_{\mathtt{m}}, \phi, \mathsf{excPostRTE}, \mathtt{m}) \ = \\ Q \Rightarrow \phi \ \begin{array}{l}[\mathbf{cntr} \backslash \mathbf{cntr} + 1] \\ [\mathbf{st}(\mathbf{cntr} + 1) \backslash v]\end{array} \\ \\ \wedge \\ R \end{array}$$

We show several cases of the proof, which is done by induction over the structure of the formula. The rest of the cases proceed in a similar way.

Proof :

1. $\mathcal{E}^{src} = const, const \in$ **constInt**, *true*, *false*

{ *source case* }
(1) $wp^{src}(const, \psi, \mathsf{excPostRTE}, \mathtt{m})_{const}$
{ *following the definition of the wp function for source expressions in subsection 7.4.2* }
$= \psi$

{ *bytecode case* }
(2) $wp^{bc}_{seq}(\ulcorner s, const, s \urcorner_{\mathtt{m}}, \phi, \mathsf{excPostRTE}, \mathtt{m})$
{ *following the definition of the compiler function in subsection 7.3.3* }
$= wp^{bc}(s \text{ push}const, \phi, \ulcorner \mathsf{excPostRTE} \urcorner, \mathtt{m})$
{ *following the definition of the wp function for bytecode in subsection 7.4.2* }
$= \phi \begin{array}{l}[\mathbf{cntr} \backslash \mathbf{cntr} + 1] \\ [\mathbf{st}(\mathbf{cntr} + 1) \backslash const]\end{array}$

{ *from (1) and (2) and* $Q, R = \mathbf{true}$ *this case holds* }

2. $\mathcal{E}^{src} = \mathcal{E}^{src}.f$

{ *source case* }
(1) $wp^{src}(\mathcal{E}^{src}.f, \psi, \mathsf{excPostRTE}, \mathtt{m})_{v.f}$
{*following the definition of the wp function for source expressions in subsection 7.4.2* }
$= wp^{src}(\mathcal{E}^{src}, \begin{array}{l} v \neq \mathbf{null} \Rightarrow \psi \\ \wedge \\ v = \mathbf{null} \Rightarrow \mathtt{m}.\mathsf{excPostRTE}(\ \mathtt{NullPntrExc}) \end{array}, \mathsf{excPostRTE}, \mathtt{m})_v$

{ *bytecode case* }
(2) $wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{src}.f, e \urcorner_{\mathtt{m}}, \phi, \ulcorner \mathsf{excPostRTE} \urcorner, \mathtt{m})$
{ *following the definition of the compiler function in subsection 7.3.3* }
$= wp^{bc}_{seq}(\begin{array}{l}\ulcorner s, \mathcal{E}^{src}, e - 1 \urcorner_{\mathtt{m}} \\ e \text{ getfield } f\end{array}, \phi, \mathsf{excPostRTE}, \mathtt{m})$
{*following the definition of the wp function for bytecode in subsection 7.5* }
$= wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{src}, e - 1 \urcorner_{\mathtt{m}},$
$\quad \mathbf{st}(\mathbf{cntr}) \neq \mathbf{null} \Rightarrow$
$\quad \phi[\mathbf{st}(\mathbf{cntr}) \backslash \mathbf{st}(\mathbf{cntr}).f]$
$\quad \wedge$
$\quad \mathbf{st}(\mathbf{cntr}) = \mathbf{null} \Rightarrow \mathtt{m}.\mathsf{excPostRTE}(\ \mathtt{NullPntrExc})$,
$\ulcorner \mathsf{excPostRTE} \urcorner, \mathtt{m})$

$\{$ *From (1) and (2) we apply the induction hypothesis* $\}$
$\exists Q', R' : \mathcal{F},$

$$\textit{(3) } wp^{src}(\mathcal{E}^{src}, \begin{array}{l} v \neq \textbf{null} \Rightarrow \psi \\ \wedge \\ v = \textbf{null} \Rightarrow \texttt{m.excPostRTE(NullPntrExc)} \end{array}, \mathsf{excPostRTE}, \texttt{m})_v$$

$=$

$$Q' \Rightarrow \begin{array}{l} v \neq \textbf{null} \Rightarrow \psi \\ \wedge \\ v \neq \textbf{null} \Rightarrow \texttt{m.excPostRTE(NullPntrExc)} \end{array} \wedge R'$$

$\Longrightarrow$

$$wp^{bc}_{seq}(\ulcorner \mathcal{E}^{src} \urcorner, \begin{array}{l} \textbf{st}(\textbf{cntr}) \neq \textbf{null} \Rightarrow \\ \phi[\textbf{st}(\textbf{cntr}) \backslash \textbf{st}(\textbf{cntr}).f] \\ \wedge \\ \textbf{st}(\textbf{cntr}) = \textbf{null} \Rightarrow \texttt{m.excPostRTE(NullPntrExc)} \end{array}, \ulcorner \mathsf{excPostRTE} \urcorner, \texttt{m})$$

$=$

$$Q' \Rightarrow \begin{array}{l} \textbf{st}(\textbf{cntr}) \neq \textbf{null} \Rightarrow \phi[\textbf{st}(\textbf{cntr}) \backslash \textbf{st}(\textbf{cntr}).f] \\ \wedge \\ \textbf{st}(\textbf{cntr}) = \textbf{null} \Rightarrow \texttt{m.excPostRTE(NullPntrExc)} \end{array} \begin{array}{l} [\textbf{cntr} \backslash \textbf{cntr} + 1] \\ [\textbf{st}(\textbf{cntr} + 1) \backslash v] \end{array} \wedge R'$$

$=$

$\{$ *ϕ contains only stack expressions* $\textbf{st}(\textbf{cntr} - k), k \geq 0$ *and properties of substitution* $\}$

$$Q' \Rightarrow \begin{array}{l} v \neq \textbf{null} \Rightarrow \phi \begin{array}{l} [\textbf{cntr} \backslash \textbf{cntr} + 1] \\ [\textbf{st}(\textbf{cntr} + 1) \backslash v.f] \end{array} \\ \wedge \\ v = \textbf{null} \Rightarrow \texttt{m.excPostRTE(NullPntrExc)} \end{array} \wedge R'$$

$\{$ *from (3) this case holds* $\}$

actually, here need an assumption that the modifies clauses cannot contain stack expressions

Lemma .2.2 (Wp of a compiled expression) *For any expression $\mathcal{E}^{src}$ from our source language, for any formula $\psi : \mathcal{F}$ of the source assertion language and any formula $\phi : P$ such that ϕ may only contain stack expressions of the form $\textbf{st}(\textbf{cntr} - k), k \geq 0$ and if the modifies clauses of every method does not contain stack expressions, then the following holds :*

there exist $Q, R : \mathcal{F}, v_1 \dots v_k : \mathcal{E}^{src}$

$$
\begin{array}{l}
wp^{src}(\mathcal{E}^{src}, \psi, \mathsf{excPostRTE}, \mathtt{m})_v = \\
\forall \mathbf{bv}, \mathbf{bv}_1, \dots \mathbf{bv}_k, Q \Rightarrow \\
\qquad \begin{array}{ll} & [v \backslash \mathbf{bv}] \\ \psi & [v_1 \backslash \mathbf{bv}_1] \\ & \dots \\ & [v_k \backslash \mathbf{bv}_k] \end{array} \\
\wedge \\
R \\
\Longrightarrow
\end{array}
$$

$$
\begin{array}{l}
wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{src}, e \urcorner_{\mathtt{m}}, \phi, \mathsf{excPostRTE}, \mathtt{m}) \; = \\
\forall \mathbf{bv}, \mathbf{bv}_1, \dots \mathbf{bv}_k, Q \Rightarrow \\
\qquad \begin{array}{ll} & [\mathbf{cntr} \backslash \mathbf{cntr} + 1] \\ & [\mathbf{st}(\mathbf{cntr} + 1) \backslash \mathbf{bv}] \\ \phi & [v_1 \backslash \mathbf{bv}_1] \\ & \dots \\ & [v_k \backslash \mathbf{bv}_k] \end{array} \\
\wedge \\
R
\end{array}
$$

This lemma refers to the cases for method invokation and instance creation. In the following, we scatch the case for instance creation but we ommit the part of the formulas concerning the exceptional termination of the evaluation of the expression.
Proof:

{ *following the definition from Fig.7.11 we get for the source case* }

(1) $wp^{src}(\textbf{new } \texttt{Cl}(\mathcal{E}^{src}), \psi, \mathsf{excPost}^{src}, \texttt{m})_v =$
$\forall \textbf{bv},$
$\neg\ \textbf{instances}(\textbf{bv}) \wedge$
$\textbf{bv} \neq \textbf{null}$
$\backslash\textbf{typeof(bv)} <: \texttt{Cl} \Rightarrow$
$wp^{src}(\mathcal{E}^{src},$

$$\left\{\begin{array}{l} \mathsf{constr}(\texttt{Cl}).\mathsf{Pre}^{src}\ [\textbf{this}\backslash\textbf{bv}]\ [arg\backslash v']\ [f\backslash f[\oplus\textbf{bv} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\mathsf{subtype}(f.\mathsf{declaredIn},\texttt{Cl})} \\ \wedge \\ \forall\ m \in \mathsf{constr}(\texttt{Cl}).\mathsf{modif}^{src}, \\ \quad \mathsf{constr}(\texttt{Cl}).\mathsf{nPost}^{src}\ [\textbf{this}\backslash\textbf{bv}]\ [arg\backslash v'][f\backslash f[\oplus\textbf{bv} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\mathsf{subtype}(f.\mathsf{declaredIn},\texttt{Cl})} \\ \quad\quad \Rightarrow \psi[v\backslash\textbf{bv}] \end{array}\right.$$

$\mathsf{excPost}^{src}, \texttt{m})_{v'}$

{ *compiler definition from Fig. 7.3 and definition of* wp^{bc}_{seq} }

(2) $wp^{bc}_{seq}(\ulcorner s, \textbf{new } \texttt{Cl}(\mathcal{E}^{src}), e\urcorner_{\texttt{m}}, \phi, \mathsf{excPostRTE}, \texttt{m}) =$

$$wp^{bc}_{seq}\left(\begin{array}{l} s : \text{new } \texttt{Cl}; \\ s+1 : \text{dup}; \\ \ulcorner s+2, \mathcal{E}^{src}, e-1\urcorner_{\texttt{m}}; \\ e : \text{invoke } \mathsf{constr}(\texttt{Cl}); \end{array}\right.,$$
$\phi,$
$\mathsf{excPostRTE}, \texttt{m}) =$

{ *definition of* wp^{bc}_{seq} }

$$wp^{bc}_{seq}\left(\begin{array}{l} s : \text{new } \texttt{Cl}; \\ s+1 : \text{dup}; \\ \ulcorner s+2, \mathcal{E}^{src}, e-1\urcorner_{\texttt{m}}; \end{array}\right.,$$
$wp^{bc}(e : \text{invoke } \mathsf{constr}(\texttt{Cl}), \phi, \mathsf{excPostRTE}, \texttt{m}),$
$\mathsf{excPostRTE}, \texttt{m}) =$

{ *definition of* wp^{bc} *for* invoke}

$$wp^{bc}_{seq}\left(\begin{array}{l} s : \text{new } \texttt{Cl}; \\ s+1 : \text{dup}; \end{array} , \right.$$

$$\quad wp^{bc}_{seq}(\ulcorner s+2, \mathcal{E}^{src}, e-1 \urcorner_{\texttt{m}};,$$

$$\qquad \mathsf{constr}(\texttt{Cl}).\mathsf{pre} \begin{array}{l} [\mathbf{reg}(0)\backslash \mathbf{st}(\mathbf{cntr}-1)] \\ [\mathbf{reg}(1)\backslash \mathbf{st}(\mathbf{cntr})] \end{array}$$

$$\qquad \wedge$$

$$\qquad \forall mod, (mod \in \mathsf{constr}(\texttt{Cl}).\mathsf{modif})(,$$

$$\qquad\quad \mathsf{constr}(\texttt{Cl}).\mathsf{normalPost} \begin{array}{l} [\mathbf{reg}(0)\backslash \mathbf{st}(\mathbf{cntr}-1)] \\ [\mathbf{reg}(1)\backslash \mathbf{st}(\mathbf{cntr})] \end{array} \Rightarrow \phi[\mathbf{cntr}\backslash \mathbf{cntr}-2])$$

$$\qquad \mathsf{excPostRTE}, \texttt{m}),$$

$$\quad \mathsf{excPostRTE}, \texttt{m})$$

{ *we assume for* $\mathcal{E}^{src}$, ψ *and* excPost^{src} *that the following holds* }

(3) $for\ some\ Q, R : \mathcal{F}$,

$$wp^{src}(\mathcal{E}^{src},$$

$$\quad \mathsf{constr}(\texttt{Cl}).\mathsf{Pre}^{src} \begin{array}{l} [\mathbf{this}\backslash \mathbf{bv}] \\ [arg\backslash v'] \\ [f\backslash f[\oplus \mathbf{bv} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f:\mathbf{Field}.\mathsf{subtype}(f.\mathsf{declaredIn},\texttt{Cl})} \end{array}$$

$$\quad \forall\ m \in \mathsf{constr}(\texttt{Cl}).\mathsf{modif}^{src},$$

$$\qquad \mathsf{constr}(\texttt{Cl}).\mathsf{nPost}^{src} \begin{array}{l} [\mathbf{this}\backslash \mathbf{bv}] \\ [arg\backslash v'] \\ [f\backslash f[\oplus \mathbf{bv} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f:\mathbf{Field}.\mathsf{subtype}(f.\mathsf{declaredIn},\texttt{Cl})} \end{array} \Rightarrow ,$$

$$\qquad \psi \begin{array}{l} [v\backslash \mathbf{bv}] \\ [f\backslash f[\oplus \mathbf{bv} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f:\mathbf{Field}.\mathsf{subtype}(f.\mathsf{declaredIn},\texttt{Cl})} \end{array}$$

$$\quad \mathsf{excPost}^{src}, \texttt{m})_{v'} =$$

$$Q \Rightarrow \left\{ \begin{array}{l} \mathsf{constr}(\texttt{Cl}).\mathsf{Pre}^{src} \begin{array}{l} [\mathbf{this}\backslash \mathbf{bv}] \\ [arg\backslash v'] \\ [f\backslash f[\oplus \mathbf{bv} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f:\mathbf{Field}.\mathsf{subtype}(f.\mathsf{declaredIn},\texttt{Cl})} \end{array} \\ \forall\ m \in \mathsf{constr}(\texttt{Cl}).\mathsf{modif}^{src}, \\ \qquad \mathsf{constr}(\texttt{Cl}).\mathsf{nPost}^{src} \begin{array}{l} [\mathbf{this}\backslash \mathbf{bv}] \\ [arg\backslash v'] \\ [f\backslash f[\oplus \mathbf{bv} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f:\mathbf{Field}.\mathsf{subtype}(f.\mathsf{declaredIn},\texttt{Cl})} \end{array} \Rightarrow \psi[v\backslash \mathbf{bv}] \end{array} \right.$$

$$\wedge$$

$$R$$

{ *which means that* }

(1) $=$

$$\forall \mathbf{bv},$$

$$\neg\ \mathbf{instances}(\mathbf{bv}) \wedge$$

$$\mathbf{bv} \neq \mathbf{null}$$

$$\backslash\mathbf{typeof(bv)} <: \texttt{Cl} \Rightarrow$$

$$Q \Rightarrow \left\{ \begin{array}{l} \mathsf{constr}(\texttt{Cl}).\mathsf{Pre}^{src} \begin{array}{l} [\mathbf{this}\backslash \mathbf{bv}] \\ [arg\backslash v'] \\ [f\backslash f[\oplus \mathbf{bv} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f:\mathbf{Field}.\mathsf{subtype}(f.\mathsf{declaredIn},\texttt{Cl})} \end{array} \\ \forall\ m \in \mathsf{constr}(\texttt{Cl}).\mathsf{modif}^{src}, \\ \qquad \mathsf{constr}(\texttt{Cl}).\mathsf{nPost}^{src} \begin{array}{l} [\mathbf{this}\backslash \mathbf{bv}] \\ [arg\backslash v'] \\ [f\backslash f[\oplus \mathbf{bv} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\forall f:\mathbf{Field}.\mathsf{subtype}(f.\mathsf{declaredIn},\texttt{Cl})} \end{array} \Rightarrow \psi[v\backslash \mathbf{bv}] \end{array} \right.$$

$$\wedge$$

$$R$$

{ *from Lemma 7.6.2 then we get* }

$$
\begin{array}{l}
\textit{(4)}\ wp^{bc}_{seq}(\ulcorner s+2, \mathcal{E}^{src}, e-1 \urcorner_{\mathtt{m}};, \\
\qquad \mathsf{constr}(\mathtt{Cl}).\mathsf{pre}\ \begin{matrix}[\mathbf{reg}(0)\backslash\mathbf{st}(\mathbf{cntr}-1)] \\ [\mathbf{reg}(1)\backslash\mathbf{st}(\mathbf{cntr})]\end{matrix} \\
\qquad \wedge \\
\qquad\ \forall mod, (mod \in \mathsf{constr}(\mathtt{Cl}).\mathsf{modif})(\\
\qquad\qquad \mathsf{constr}(\mathtt{Cl}).\mathsf{normalPost}\ \begin{matrix}[\mathbf{reg}(0)\backslash\mathbf{st}(\mathbf{cntr}-1)] \\ [\mathbf{reg}(1)\backslash\mathbf{st}(\mathbf{cntr})]\end{matrix} \Rightarrow \phi[\mathbf{cntr}\backslash\mathbf{cntr}-2]) \\
\qquad \mathsf{excPostRTE}, \mathtt{m}) =
\end{array}
,
$$

$$
Q \Rightarrow \left\{ \begin{array}{l}
\mathsf{constr}(\mathtt{Cl}).\mathsf{pre}\ \begin{matrix}[\mathbf{reg}(0)\backslash\mathbf{st}(\mathbf{cntr}-1)] \\ [\mathbf{reg}(1)\backslash\mathbf{st}(\mathbf{cntr})]\end{matrix} \\
\wedge \\
\ \forall mod, (mod \in \mathsf{constr}(\mathtt{Cl}).\mathsf{modif}) \\
\qquad \mathsf{constr}(\mathtt{Cl}).\mathsf{normalPost}\ \begin{matrix}[\mathbf{reg}(0)\backslash\mathbf{st}(\mathbf{cntr}-1)] \\ [\mathbf{reg}(1)\backslash\mathbf{st}(\mathbf{cntr})]\end{matrix} \\
\qquad \Rightarrow \phi[\mathbf{cntr}\backslash\mathbf{cntr}-2]
\end{array} \right\} \begin{matrix}[\mathbf{cntr}\backslash\mathbf{cntr}+1] \\ [\mathbf{st}(\mathbf{cntr}+1)\backslash v']\end{matrix}
$$

$\wedge$
R
$=$

{*apply substitution and because* $\mathsf{constr}(\mathtt{Cl})$ .modif *does not contain stack expressions and because of the initial hypothesis about* ϕ *obtain* }

$$
Q \Rightarrow \left\{ \begin{array}{l}
\mathsf{constr}(\mathtt{Cl}).\mathsf{pre}\ \begin{matrix}[\mathbf{reg}(0)\backslash\mathbf{st}(\mathbf{cntr})] \\ [\mathbf{reg}(1)\backslash v']\end{matrix} \\
\wedge \\
\ \forall mod, (mod \in \mathsf{constr}(\mathtt{Cl}).\mathsf{modif})(\\
\qquad \mathsf{constr}(\mathtt{Cl}).\mathsf{normalPost}\ \begin{matrix}[\mathbf{reg}(0)\backslash\mathbf{st}(\mathbf{cntr})] \\ [\mathbf{reg}(1)\backslash v']\end{matrix} \\
\qquad \Rightarrow \phi[\mathbf{cntr}\backslash\mathbf{cntr}-1])
\end{array} \right.
$$

$\wedge$
R

{ *from (2) and (4)* }

(5) $wp^{bc}_{seq}(\ulcorner s, \textbf{new } \texttt{Cl}(\mathcal{E}^{src}), e\urcorner_{\texttt{m}}, \phi, \mathsf{excPostRTE}, \texttt{m}) =$
$wp^{bc}_{seq}($ s : new Cl;
$s+1$: dup; ,

$$Q \Rightarrow \left\{ \begin{array}{l} \mathsf{constr}(\texttt{Cl}).\mathsf{pre}\ [\mathbf{reg}(0)\backslash\mathbf{st}(\mathbf{cntr})][\mathbf{reg}(1)\backslash v'] \\ \wedge \\ \forall mod, (mod \in \mathsf{constr}(\texttt{Cl}).\mathsf{modif})(\\ \quad \mathsf{constr}(\texttt{Cl}).\mathsf{normalPost}\ [\mathbf{reg}(0)\backslash\mathbf{st}(\mathbf{cntr})][\mathbf{reg}(1)\backslash v'] \\ \quad \Rightarrow \phi[\mathbf{cntr}\backslash\mathbf{cntr}-1]) \end{array} \right. ,$$

$\mathsf{excPostRTE}, \texttt{m}) =$

{ *apply rule* wp^{bc} *for* dup}

$wp^{bc}_{seq}(\ulcorner s, \textbf{new } \texttt{Cl}(\mathcal{E}^{src}), e\urcorner_{\texttt{m}}, \phi, \mathsf{excPostRTE}, \texttt{m}) =$
$wp^{bc}_{seq}(s$: new Cl;
,

$$Q \Rightarrow \left\{ \begin{array}{l} \mathsf{constr}(\texttt{Cl}).\mathsf{pre}\ [\mathbf{reg}(0)\backslash\mathbf{st}(\mathbf{cntr})][\mathbf{reg}(1)\backslash v'] \\ \wedge \\ \forall mod, (mod \in \mathsf{constr}(\texttt{Cl}).\mathsf{modif}) \\ \quad \mathsf{constr}(\texttt{Cl}).\mathsf{normalPost}\ [\mathbf{reg}(0)\backslash\mathbf{st}(\mathbf{cntr})][\mathbf{reg}(1)\backslash v'] \\ \quad \Rightarrow \phi \end{array} \right. ,$$

$\mathsf{excPostRTE}, \texttt{m}) =$

{ *apply rule* wp^{bc} *for* new}

$$\begin{array}{l} \forall \mathbf{bv}, \\ \quad \neg\ \mathbf{instances}(\mathbf{bv}) \wedge \\ \quad \mathbf{bv} \neq \mathbf{null} \\ \quad \backslash\mathbf{typeof(bv)} <: \texttt{Cl} \Rightarrow \\ \quad Q \Rightarrow \left\{ \begin{array}{l} \mathsf{constr}(\texttt{Cl}).\mathsf{pre}\ [\mathbf{reg}(0)\backslash\mathbf{bv}][\mathbf{reg}(1)\backslash v'][f\backslash f[\oplus\mathbf{bv} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\mathsf{subtype}(f.\mathsf{declaredIn},\texttt{Cl})} \\ \wedge \\ \forall mod, (mod \in \mathsf{constr}(\texttt{Cl}).\mathsf{modif}) \\ \quad \mathsf{constr}(\texttt{Cl}).\mathsf{normalPost}\ [\mathbf{reg}(0)\backslash\mathbf{bv}][\mathbf{reg}(1)\backslash v'][f\backslash f[\oplus\mathbf{bv} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\mathsf{subtype}(f.\mathsf{declaredIn},\texttt{Cl})} \\ \quad \Rightarrow \phi \\ [\mathbf{cntr}\backslash\mathbf{cntr}+1] \\ [\mathbf{st}(\mathbf{cntr}+1)\backslash\mathbf{bv}] \\ [f\backslash f[\oplus\mathbf{bv} \rightarrow \mathsf{defVal}(f.\mathsf{Type})]]_{\mathsf{subtype}(f.\mathsf{declaredIn},\texttt{Cl})} \end{array} \right. \end{array}$$

{ *from the equalities (5) , (1), (2) and assumption (3) this case holds* }

Qed.

Bibliography

[1] Java In The Small. http://www.lifl.fr/RD2P/JITS/.

[2] JC Virtual Machine. http://jcvm.sourceforge.net/.

[3] R.M. Amadio, S. Coupet-Grimal, S. Dal Zilio, and L. Jakubiec. A Functional Scenario for Bytecode Verification of Resource Bounds. Research report 17-2004, LIF, Marseille, France, 2004.

[4] D. Aspinall and A. Compagnoni. Heap-bounded assembly language. *J. Autom. Reason.*, 31(3-4):261–302, 2003.

[5] D. Aspinall and M. Hofmann. Another type system for in-place update. In *ESOP*, volume 2305 of *LNCS*, pages 36–52, 2002.

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

[7] J. Ullman A.V. Aho, R. Sethi. *Compilers Principles, Techniques and Tools*. Addison-Wesley Publiching Company, 1986.

[8] Ana Azevedo, Alex Nicolau, and Joe Hummel. Java annotation-aware just-in-time (ajit) complilation system. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 142–151, New York, NY, USA, 1999. ACM Press.

[9] Fabian Bannwart. A logic for bytecode and the translation of proofs from sequential java. Technical report, ETHZ, 2004.

[10] Fabian Bannwart and Peter Muller. A program logic for bytecode. In *Bytecode 2005*, ENTCS, 2005.

[11] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. In "G.Barthe, L.Burdy, M.Huisman, J.Lanet, and T.Muntean", editors, *CASSIS workshop proceedings*, LNCS, pages 49–69. Springer, 2004.

[12] G. Barthe, T. Rezk, and A. Saabas. Proof obligations preserving compilation. pages 112–126, 2005.

[13] 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.

[14] 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.

[15] Nick Benton. A typed logic for stack and jumps. DRAFT, 2004.

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

[17] C. Breunesse, N. Cataño, M. Huisman, and B. Jacobs. Formal methods for smart cards: an experience report. *Science of Computer Programming*, 2004. To appear.

[18] L. Burdy. A treatment of partiality: Its application to the b method, 1998.

[19] 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.

[20] 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.

[21] D. Cachera, T. Jensen, D. Pichardie, and G. Schneider. Certified memory usage analysis. Submitted, 2005.

[22] A. Chander, D. Espinosa, N. Islam, P. Lee, and G. Necula. Enforcing resource bounds via static verification of dynamic checks. To appear, 2005.

[23] 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.

[24] K. Crary and S. Weirich. Resource bound certification. In *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 184–198. ACM Press, 2000.

[25] Marc Daumas, Laurence Rideau, and Laurent Thry. A generic library for floating-point numbers and its application to exact computing. page 169, 2003.

[26] Draft Revision December. Jml reference manual.

[27] Edsger W. Dijkstra and Carel S. Scholten. *Predicate Calculus and Program Semantics*. Springer, 1990.

[28] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.

[29] L.-A. Fredlund. Guaranteeing correctness properties of a java card applet. In *RV'04*, ENTCS, 2004.

[30] 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.

[31] D. Garbervetsky, C. Nakhli, S. Yovine, and H. Zorgati. Program instrumentation and run-time analysis of scoped memory in java. In *RV'04*, ENCS, 2004.

[32] David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, number 1000, pages 366–373. Springer-Verlag, New York, N.Y., 1995.

[33] G. Grimaud and J.-J. Vandewalle. Introducing research issues for next generation Java-based smart card platforms. In *Proc. Smart Objects Conference (sOc'2003)*, Grenoble, France, 2003.

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

[35] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[36] M. Hofmann. A type system for bounded space and functional in-place update. *Nordic Journal of Computing*, 7(4):258–289, 2000.

[37] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. of 30th ACM Symp. on Principles of Programming Languages (POPL'03)*, pages 185–197. ACM Press, 2003.

[38] J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: Towards embedded ML programming. In *International Conference on Functional Programming*, pages 70–81, 1999.

[39] Joseph Hummel, Ana Azevedo, David Kolson, and Alexandru Nicolau. Annotating the Java bytecodes in support of optimization. *Concurrency: Practice and Experience*, 9(11):1003–1016, 1997.

[40] Kazuaki Ishizaki, Motohiro Kawahito, Toshiaki Yasue, Mikio Takeuchi, Takeshi Ogasawara, Toshio Suganuma, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Design, implementation, and evaluation of

optimizations in a just-in-time compiler. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 119–128, New York, NY, USA, 1999. ACM Press.

[41] B. Jacobs and E. Poll. Java program verification at nijmegen: Developments and perspective, 2003.

[42] 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.

[43] Laurent Lagosanto. Next-generation embedded java operating system for smart cards. In *4th Gemplus Developer Conference*, 2002.

[44] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[45] "K. Rustan M. Leino, Greg Nelson, , and James B. Saxe ". Esc/java user's manual.

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

[47] Xavier Leroy. Java bytecode verification: Algorithms and formalizations. In *Journal of Automated Reasoning 2003*, 2003.

[48] Tim Lindholm and Frank Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., 1999.

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

[50] C. Marche, C. Paulin-Mohring, and X. Urbain. The krakatoa tool for certification of java/javacard programs annotated in jml, 2003.

[51] M.Barnett and K. Rustan M. Leino. Weakest-precondition of unstructured programs. Microsoft Research, One Microsoft Way, Redmond, WA 98052, USA.

[52] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.

[53] Deepak Mulchandani. Java for embedded systems. *Internet Computing, IEEE*, 2(3):30 – 39, 1998.

[54] Gilles Muller, Barbara Moura, Fabrice Bellard, and Charles Consel. Harissa: a flexible and efficient java environment mixing bytecode and compiled code. In *Third USENIX Conference on Object-Oriented Technologies (COOTS)*. USENIX, Portland, Oregon, June 1997.

[55] George Necula. *Compiling With Proofs*. PhD thesis, Carnegie Mellon University, 1998. Phd thesis.

[56] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *PLDI98*, 1998.

[57] Tobias Nipkow. Hoare logics in Isabelle/HOL. In H. Schwichtenberg and R. Steinbrüggen, editors, *Proof and System-Reliability*, pages 341–367. Kluwer, 2002.

[58] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson. Toba: Java for applications: A way ahead of time (wat) compiler. In *Third USENIX Conference on Object-Oriented Technologies (COOTS)*, Portland, Oregon, June 1997. University of Arizona.

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

[60] 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.

[61] C.L. Quigley. A programming logic for Java bytecode programs. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, volume 2758 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[62] A.D. Raghavan and G.T. Leavens. Desugaring JML method specification. Report 00-03d, Iowa State University, Department of Computer Science, 2003.

[63] R.W.Floyd. Assigning meaning to programs. In J. T. Schwartz, editor, *volume 19 of Proceedings of Symposia in Applied Mathematics*, pages 19–32, 1967.

[64] Ando Saabas and Tarmo Uustalu. A compositional natural semantics and hoare logic for low-level languages. In *Conf. version in P. D Mosses, I. Ulidowski, eds., Proc. of 2nd Wksh. on Structured Operational Semantics, SOS'05 (Lisbon, July 2005)*, pages 151–168, 2005.

[65] Birgit Schieder and Manfred Broy. Adapting calculational logic to the undefined. *The Computer Journal*, 42(2):73–81, ???? 1999.

[66] G. Schneider. A constraint-based algorithm for analysing memory usage on java cards. Technical Report RR-5440, INRIA, December 2004.

[67] I. Siveroni. Operational semantics of the java card virtual machine, 2004.

[68] Raymie Stata and Martín Abadi. A type system for Java bytecode subroutines. In *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, California*, pages 149–160, New York, NY, 1998.

[69] Raja Vallée-Rai, Laurie Hendren, Vijay Sundaresan, Patrick Lam, Etienne Gagnon, and Phong Co. Soot - a java optimization framework. In *Proceedings of CASCON 1999*, pages 125–135, 1999.

[70] J. C. Vanderwaart and K. Crary. Foundational typed assembly language for grid computing. Technical Report CMU-CS-04-104, CMU, February 2004.