Compiling Proof Obligations

Mariela Pavlova

July 11, 2006

Contents

1   Introduction

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 **??**. 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 **??**) 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 1.1 presents an overview of existing work on this subject. In Section 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 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 4 presents the weakest precondition predicate transformer which we define over the source language.

Section 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 **??** and this new version are equivalent.

In Section 6, we proceed with the proof of equivalence between the proof obligations generated by the weakest precondition defined in Chapter **??**. 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 also the equivalence between the two

bytecode weakest preconditions to conclude that the source verification condition generator and the bytecode verification condition generator presented in Chapter **??**

1.1 Related Work

Several works dealing with the relation between source and its compilation verification condition generated.

Barthe, Rezk and Saabas in **??** also argue that proof obligations produced over source code and bytecode produced by a nonoptimizing compiler are equivalent. The source language supports method invokation, exception throwing and handling.They do not consider instructions that may throw runtime exceptions. Note that because of this

However, in the article they do not discuss at all what are the assumptions or properties of the compiler which will guarantee this equivalence. We claim that their proof for the verification condition preserving compilation holds only if their non optimizing compiler has the properties discussed here in Section 3.

because they do not expose those conditions, they cannot use the induction hypothesis because they have not proven that the condition to apply the induction hold

2 Source language

We present a source Java-like programming language which supports the following features: object manipulation and creation, method invocation, throwing and handling exceptions, subroutines etc. The first definition that we give hereafter presents all the constructs of our language which evaluate to a value.

Definition 2.1 (Expression) *The grammar for source expressions is defined as follows*

$$
\begin{aligned}
\mathcal{E}^{src} ::= \quad & \textbf{constInt} \\
& | \ \textbf{true} \\
& | \ \textbf{false} \\
& | \ \mathcal{E}^{src} \ \textbf{op} \ \mathcal{E}^{src} \\
& | \ \mathcal{E}^{src}.f \\
& | \ \textbf{var} \\
& | \ (Class) \ \mathcal{E}^{src} \\
& | \ \textbf{null} \\
& | \ \textbf{this} \\
& | \ \mathcal{E}^{\mathcal{R}} \\
& | \ \mathcal{E}^{src}.m() \\
& | \ \textbf{new} \ Class(\mathcal{E}^{src})
\end{aligned}
$$

$$
\begin{aligned}
\mathcal{E}^{\mathcal{R}} ::= \quad & \mathcal{E}^{src} \ \mathcal{R} \ \mathcal{E}^{src} \\
& | \ \mathcal{E}^{src} \ \textbf{instanceof} \ Class
\end{aligned}
$$

$$
\mathcal{R} \in \{\leq, <, \geq, >, =, \neq\}
$$

We now a give an informal description of the meaning of the expressions of the above grammar:

- **constInt** is any integer literal

3

- **true** and **false** are the unique boolean constants

- **constRef** is a reference to an object in the memory heap

- $\mathcal{E}^{src}$ *op* $\mathcal{E}^{src}$ which stands for an arithmetic expression with any of the arithmetic operators $+, -, div, rem, *$

- $\mathcal{E}^{src}.f$ is a field access expression where the field with name f is accessed

- the cast expression $(Class)\mathcal{E}^{src}$ which is applied only to expressions from a reference type

- the expression **null** stands for the null reference which does not point to any location in the heap

- **this** refers to the current object

- $\mathcal{E}^{src}.m()$ stands for a method invokation expression. Note that here we consider only methods without arguments which return a value

- **new** $Class(\mathcal{E}^{src})$ stands for an object creation expression of class $Class$. We consider only constructors which take only one argument for the sake of readability

The language is also provided with relational expressions, which evaluate to the boolean values:

- $\mathcal{E}^{src} \mathcal{R} \mathcal{E}^{src}$ where $\mathcal{R} \in \{\leq, <, \geq, >, =, \neq\}$ stands for the relation between two expressions

- $\mathcal{E}^{src}$ **instanceof** $Class$ states that $\mathcal{E}^{src}$ has as type the class $Class$ or one of its subclasses

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

$$\texttt{JavaType} ::= Class, \ Class \in \ \texttt{ClassTypes} \mid \texttt{int} \mid \texttt{boolean}$$

The next definition gives the control flow constructs of our language as well as the expressions that have a side effect

Definition 2.2 (Statement) *The grammar for expressions is defined as follows :*

$$
\begin{aligned}
\mathcal{STMT} ::= \ & \mathcal{STMT}; \mathcal{STMT} \\
& \mid \texttt{if } (\mathcal{E}^{\mathcal{R}}) \texttt{ then } \{\mathcal{STMT}\} \texttt{ else } \{\mathcal{STMT}\} \\
& \mid \texttt{try } \{\mathcal{STMT}\} \texttt{ catch } (\texttt{Exc }) \{\mathcal{STMT}\} \\
& \mid \texttt{try } \{\mathcal{STMT}\} \texttt{ finally } \{\mathcal{STMT}\} \\
& \mid \texttt{throw } \mathcal{E}^{src} \\
& \mid \texttt{while } (\mathcal{E}^{\mathcal{R}})[\texttt{INV}, \texttt{modif}] \ \{\mathcal{STMT}\} \\
& \mid \texttt{return } \mathcal{E}^{src} \\
& \mid \mathcal{E}^{src} = \mathcal{E}^{src}
\end{aligned}
$$

est-ce que je dois dire qu'on considere un sousensemble de Class qui represente les exceptions ?

From the definition we can see that the language supports also the following constructs :

- $\mathcal{STMT};\mathcal{STMT}$, i.e. statements that execute sequentially

- 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

- try $\{\mathcal{STMT}\}$ catch $(Class)$ $\{\mathcal{STMT}\}$ which states 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

- try $\{\mathcal{STMT}\}$ finally $\{\mathcal{STMT}\}$

- while $(\mathcal{E}^{\mathcal{R}})[\text{INV},\text{modif}]$ $\{\mathcal{STMT}\}$ states for a loop statement where the body statement $\mathcal{STMT}$ will be executed until the relational expression $\mathcal{E}^{\mathcal{R}}$ evaluates to true.

- return $\mathcal{E}^{src}$ is the statement by which the execution will be finished

- $\mathcal{E}^{src} = \mathcal{E}^{src}$ stands for an assignment expression, where the value of the left expression is updated with the value of the right expression

- finally, every expression $\mathcal{E}^{src}$ is a statement

3   Compiler

We now turn to specify a simple compiler from the source language presented in Section 2 into the bytecode language. The compiler does not perform any optimizations.

The compiler is the triple

$$< \ulcorner *, *, * \urcorner, \mathsf{addExcHandler}, \mathsf{addLoopSpec} >$$

The first element in the triple is a function $\ulcorner \urcorner$ which transforms statements and expressions into a sequence of bytecode instructions, the second element is the procedure addExcHandler which keeps track of the exception handlers. The third element of the compiler is the procedure addLoopSpec which manages the compilation of loop specification, namely loop invariants and the modified expressions.

In the following, in the next Section 3.1 we define the procedure for exception handlers, in Section 3.2, we will present the procedure for compiling loops and in Sections **??** and 3.4 we will proceed with the definition of the function $\ulcorner \urcorner$ for exressions and statements.

the excep-
tion handler
function

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 **??**) every time it sees one. To do this, the compiler is provided with a procedure with the following name and signature:

$$\text{addExcHandler} : nat \rightarrow nat \rightarrow nat \rightarrow \mathbf{Class}_{exc} \rightarrow \mathbf{Method}$$

The definition of the procedure is the following:

$$\text{addExcHandler}(start,\ end,\ h,\ \texttt{Exc},\texttt{m})) =$$
$$\texttt{m}.\text{excHndlS} := \{(start, end, h, \texttt{Exc}), \texttt{m}.\text{excHndlS}\}$$

The function adds a new element in the exception handler table of a method $\texttt{m}$. The meaning of the new element is that every exception of type $\texttt{Exc}$ thrown in between the instructions $start \dots end$ can be handled by the code starting at index h.

We can remark that when the function addExcHandler adds a new element in the exception handler table array of a method $\texttt{m}$, the new element is added at the beginning of the exception handler table $\texttt{m}.\text{excHndlS}$.

3.2   Compiling loop invariants

When compiling a method $\texttt{m}$, the compiler will also take care of the loop specification in the source loops by adding it in the loop specification table $\texttt{m}.\text{loopSpecS}$(defined in Section **??**) of the method $\texttt{m}$.

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

$$\text{addLoopSpec} : nat \rightarrow \mathcal{F}^{bc} \rightarrow list\ \mathcal{E} \rightarrow \mathbf{Method}$$

The definition of the procedure is hereafter:

$$\text{addLoopSpec}(i,\ \texttt{INV},\ \texttt{modif},\texttt{m}) =$$
$$\texttt{m}.\text{loopSpecS} := \{(i, \texttt{INV}, \texttt{modif}), \texttt{m}.\text{loopSpecS}\}$$

3.3   Compiling expressions in bytecode instructions

As we stated in the beginning the function for compiling statements 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 of $\mathcal{STMT}$ starts, the statement $\mathcal{STMT}$ to be compiled and a natural number which is the greatest label in the compilation of $\mathcal{STMT}$ and returns an array of bytecode instructions. In the next we look at the definition of the compiler over expressions.

- integer or boolean constant access

   - integer constant access

$$\ulcorner s, \mathbf{constInt}, s \urcorner = s : \ \text{push } \mathbf{constInt}$$

   - boolean constant access

$$\ulcorner s, \mathbf{true}, s \urcorner = s : \ \text{push } 1$$

$$\ulcorner s, \mathbf{false}, s \urcorner = s : \ \text{push } 0$$

Note: the source boolean expressions are compiled down to integers

- method invokation

$$\ulcorner s, \mathcal{E}_1^{src}.m(\mathcal{E}_2^{src}), e \urcorner = \begin{array}{l} \ulcorner s, \mathcal{E}_1^{src}, e' \urcorner; \\ \ulcorner e'+1, \mathcal{E}_2^{src}, e-1 \urcorner; \\ e: \ \text{invoke } m \end{array}$$

- field access

$$\ulcorner s, \mathcal{E}^{src}.f, e \urcorner = \begin{array}{l} \ulcorner s, \mathcal{E}^{src}, e-1 \urcorner; \\ e: \ \text{getfield } f \end{array}$$

- local variable access

$$\ulcorner s, \textbf{var}, s \urcorner = s: \ \text{load } \texttt{reg}_\texttt{i}$$

where $\texttt{reg}_\texttt{i}$ is the local variable at index i

- arithmetic expressions

$$\ulcorner s, \mathcal{E}_1^{src} \ op \ \mathcal{E}_2^{src}, e \urcorner = \begin{array}{l} \ulcorner s, \mathcal{E}_1^{src}, e' \urcorner; \\ \ulcorner e'+1, \mathcal{E}_2^{src}, e-1 \urcorner; \\ e: \ \text{arith\_op} \end{array}$$

- cast expression

$$\ulcorner s, (\ \texttt{Class})\ \mathcal{E}^{src}, e \urcorner = \begin{array}{l} \ulcorner s, \mathcal{E}^{src}, e-1 \urcorner; \\ e: \ \text{checkcast } \texttt{Class} \ ; \end{array}$$

- instanceof expression

$$\ulcorner s, \mathcal{E}^{src} \ \textbf{instanceof} \ Class, e \urcorner = \begin{array}{l} \ulcorner s, \mathcal{E}^{src}, e-1 \urcorner; \\ e: \ \text{instanceof } Class; \end{array}$$

- null expression

$$\ulcorner s, \textbf{null}, s \urcorner = s: \ \text{push } \textbf{null}$$

- object creation

$$\ulcorner s, \textbf{new } Class(\mathcal{E}^{src}), e \urcorner = \begin{array}{l} s: \ \text{new } Class; \\ s+1: \ \text{dup}; \\ \ulcorner s+2, \mathcal{E}^{src}, e-1 \urcorner; \\ e: \ \text{invoke } \textsf{constr}(Class); \end{array}$$

- this instance

$$\ulcorner s, \textbf{this}, s \urcorner = s: \ \text{load } \texttt{reg}_\texttt{0}$$

3.4 Compiling control statements in bytecode instructions

- compositional statement

$$\ulcorner s, \mathcal{STMT}_1; \mathcal{STMT}_2, e \urcorner = \\ \ulcorner s, \mathcal{STMT}_1, e' \urcorner; \\ \ulcorner e'+1, \mathcal{STMT}_2, e \urcorner$$

about the compilation of exception handlers

a redundant jump added in the compilation in order to see explicitly the relation between stmt1 and stmt2

7

- if statement

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

- assignment statement. We consider the case for instance field assignment as well as assignemnts to method local variables and parameters.

  - field assignement.

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

  - method local variable or parameter update

$$\ulcorner s, \mathbf{var} = \mathcal{E}^{src}, e \urcorner =$$
$$\ulcorner s, \mathcal{E}^{src}, e - 1 \urcorner$$
$$e: \texttt{ store } \texttt{reg}_i;$$

- try catch statement

$$\ulcorner s, \texttt{try } \{\mathcal{STMT}_1\} \texttt{ catch } (\texttt{ExcClass } \mathbf{var})\{\mathcal{STMT}_2\}, e \urcorner =$$
$$\ulcorner s, \mathcal{STMT}_1, e' \urcorner;$$
$$e' + 1: \texttt{ goto } e + 1;$$
$$\ulcorner e' + 2, \mathcal{STMT}_2, e \urcorner;$$

$$\textsf{addExcHandler}(s, \ e', \ e' + 2, \ \texttt{ExcClass}, \texttt{m}))$$

  The compiler compiles the normal statement $\mathcal{STMT}_1$ and the exception handler $\mathcal{STMT}_2$.

- try finally statement

$$\ulcorner s, \texttt{try } \{\mathcal{STMT}_1\} \texttt{ finally } \{\mathcal{STMT}_2\}, e \urcorner =$$
$$\ulcorner s, \mathcal{STMT}_1, e' \urcorner;$$
$$\ulcorner e' + 1, \mathcal{STMT}_2, e'' \urcorner;$$
$$e'' + 1: \texttt{ goto } e + 1;$$

$$\{ \text{ default exception handler}\}$$
$$e'' + 2: \texttt{ store } l;$$
$$\ulcorner e'' + 3, \mathcal{STMT}_2, e - 2 \urcorner;$$
$$e - 1: \texttt{ load } l;$$
$$e: \texttt{ athrow};$$

$$\textsf{addExcHandler}(s, \ e', \ e'' + 2, \ \texttt{Exception}, \texttt{m})$$

As you can notice, 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 $\ulcorner s, \mathcal{STMT}_1, e' \urcorner$. 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 $\ulcorner e'' + 3, \mathcal{STMT}_2, e - 2 \urcorner$ 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 jsr has 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 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

say the name of the compiler

- throw exception statement

$$\ulcorner s, \text{ athrow } \mathcal{E}^{src}, e \urcorner = \begin{array}{l} \ulcorner s, \mathcal{E}^{src}, e - 1 \urcorner; \\ e : \text{ athrow}; \end{array}$$

- loop statement

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

$$\texttt{addLoopSpec}(e' + 1, \ \texttt{INV}, \ \texttt{modif}, \texttt{m})$$

- return statement

$$\ulcorner s, \texttt{return } \mathcal{E}^{src}, e \urcorner = \begin{array}{l} \ulcorner s, \mathcal{E}^{src}, e - 1 \urcorner; \\ e: \ \text{return} \end{array}$$

3.5 Properties of the compiler function

In this last subsection, we will look at the properties of the compiled statements. 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 ins_e in the compilation $\ulcorner s, \mathcal{STMT}, e \urcorner$ of a statement $\mathcal{STMT}$ is always in execution relation with the instruction ins_{e+1}

Property 3.5.1 (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 $\ulcorner s, \mathcal{STMT}, e \urcorner$ such that*

$$ins_e \rightarrow ins_{e+1}$$

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

Informally, the following property states that if there are instructions inside a compiled statetement or expression $\ulcorner s, \mathcal{S}, e \urcorner$ which are in execution relation[1] with an instruction ins_k which is not the start of an exception handler and which is outside $\ulcorner s, \mathcal{S}, e \urcorner$ then $k = e + 1$. The conditions $\neg(inList(\ulcorner s, \mathcal{S}, e \urcorner, ins_k))$ and $\neg isExcHandlerStart(ins_k)$ eliminate the case when the execution relation is between an instruction inside $\ulcorner s, \mathcal{S}, e \urcorner$ which may throw an exception and the start instruction of the proper handler exception handler.

Property 3.5.2 (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 $\ulcorner s, \mathcal{S}, e \urcorner$ such that:*

$$\forall i, (inList(\ulcorner s, \mathcal{S}, e \urcorner, ins_i)) \wedge$$
$$(ins_i \rightarrow ins_k) \wedge$$
$$\neg(inList(\ulcorner s, \mathcal{S}, e \urcorner, ins_k))$$
$$\neg isExcHandlerStart(ins_k) \Rightarrow$$
$$k = e + 1 \wedge$$

---

[1] see Def. **??**

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* $\exists, s \le i \le e, \exists k, k \notin [s \dots e+1] \wedge ins_i \rightarrow ins_k$ $\}$

$\{$ *by definition of the compiler function for if statements in section* **??** $\}$
$\ulcorner s, \texttt{if } (\mathcal{E}^{\mathcal{R}}) \texttt{ then } \{\mathcal{STMT}_1\} \texttt{ else } \{\mathcal{STMT}_2\}, e \urcorner =$
$\ulcorner s, \mathcal{E}^{\mathcal{R}}, e' \urcorner;$
$e'+1$ if $e''+2;$
$\ulcorner e'+2, \mathcal{STMT}_2, e'' \urcorner$
$e''+1$ goto $e+1;$
$\ulcorner e''+2, \mathcal{STMT}_1, e \urcorner;$

(1) $\{$ *Assume that* $s \le i \le e'$ $\}$
$\{$ *by induction hypothesis for* $\mathcal{E}^{\mathcal{R}}$ *we get* $\}$
$(2) \forall i, s \le i \le e', \ ins_i \rightarrow ins_k) \wedge$
    $\neg (inList(\ulcorner s, \mathcal{STMT}, e \urcorner, ins_k))$
    $\neg isExcHandlerStart(ins_k) \Rightarrow$
        $k = e'+1$
$\{$ *From (1) and (2) we get a contradiction in this case*$\}$

(3) $\{$ *Assume that* $e'+2 \le i \le e''$ $\}$
$(4) \ \forall i, e'+2 \ge i \le e'', \ ins_i \rightarrow ins_k) \wedge$
    $\neg (inList(\ulcorner s, \mathcal{STMT}, e \urcorner, ins_k))$
    $\neg isExcHandlerStart(ins_k) \Rightarrow$
        $k = e''+1$
$\{$ *From (3) and (4) we get a contradiction in this case*$\}$

(5) $\{$ *Assume that* $e''+2 \le i \le e$ $\}$
$(6) \ \forall i, e''+2 \ge i \le e, \ ins_i \rightarrow ins_k) \wedge$
    $\neg (inList(\ulcorner s, \mathcal{STMT}, e \urcorner, ins_k))$
    $\neg isExcHandlerStart(ins_k) \Rightarrow$
        $k = e+1$

$\{$ *From (5) and (6) we get a contradiction in this case*$\}$

(7)$\{$ *Assume that* $s = e'+1$ $\}$
$\{$ *as* $e'+1$ *if* $e''+2$ *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*

Another 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 list, i.e. the control flow can reach the instructions representing the compilation $\ulcorner s, \mathcal{S}, e \urcorner$ of statement $\mathcal{S}$ only by passing through the beginning of the compilation i_s.

Property 3.5.3 (Compilation of statements and expressions) *For all statements $\mathcal{S}'$ and $\mathcal{S}$, such that*

- *$\mathcal{S}$ is such that it has as substatement $\mathcal{S}'$, which we denote with $\mathcal{S}[\mathcal{S}']$*

- *their compilations are $\ulcorner s, \mathcal{S}, e \urcorner$ and $\ulcorner s', \mathcal{S}', e' \urcorner$*

then :

$$
\begin{aligned}
\neg(&\exists i_j, \exists i_k, \\
&inList(\ulcorner s, \mathcal{S}[\mathcal{S}'], e \urcorner, i_j) \wedge \\
&\neg inList(\ulcorner s', \mathcal{S}', e' \urcorner, i_j) \wedge \\
&inList(\ulcorner s', \mathcal{S}', e' \urcorner, i_k) \wedge \\
&s' \neq k \wedge \quad i_j \rightarrow i_k)
\end{aligned}
$$

The proof is done by induction over the structure of statements and expressions and uses the previous lemma 3.5.2

The following property establishes that the compiler preserves the substatement relation between statements and expressions.

Property 3.5.4 (Substatement and subexpression relation preserved) *For all statements $\mathcal{S}_1$ and $\mathcal{S}_2$ such that their compilations are $\ulcorner s_1, \mathcal{S}_1, e_1 \urcorner$ and that $\ulcorner s_2, \mathcal{S}_2, e_2 \urcorner$, 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:*

$$
\begin{aligned}
&\ulcorner s_2, \mathcal{S}_2, e_2 \urcorner \in \ulcorner s_1, \mathcal{S}_1, e_1 \urcorner \\
&\vee \\
&\ulcorner s_1, \mathcal{S}_1, e_1 \urcorner \in \ulcorner s_2, \mathcal{S}_2, e_2 \urcorner
\end{aligned}
$$

Now, we give a definition for a set of instructions such that they execute sequentially

Definition 3.5.1 (Block of instructions) *If the list of instructions $l = [ins_{i_1} \ldots ins_{i_n}]$ in the compilation of method m is such that*

- *none of the instructions is a target of an instruction ins_{i_j} which does not belong to l except for ins_{i_1}*

- *none of the instructions in the set is a jump instruction, i.e. $\forall m, m = 1..k \Rightarrow \neg(ins_{i_m} \in \{ \text{ goto }, \text{ if}\})$*

- *$\forall j, i_1 < j \leq i_n, \neg \exists k \in \text{m.body}, ins_k \rightarrow^l ins_j$*

We denote such a list of instructions with $ins_{i_1}; \ldots; ins_{i_k}$

The next lemma states that the compilation of an expression $\mathcal{E}^{src}$ results in a block of bytecode instructions.

Property 3.5.5 (Compilation of expressions) *For any expression $\mathcal{E}^{src}$, starting label s and end label e, the compilation $\ulcorner s, \mathcal{E}^{src}, e \urcorner$ is a block of bytecode instruction in the sense of Def. 3.5.1*

Following the definition 3.5.1 of block of bytecode instructions, the property states that the compilation of an expression results in a list of instructions that cannot be jumped from outside its compilation except for the first instruction of the compilation. This follows from lemma 3.5.3.

Definition 3.5.1 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. 3.5.1 states that the compilation $\ulcorner s, \mathcal{E}^{src}, e \urcorner$ is such that no instruction except ins_s may be a loop entry in the sense of Def. **??** in Chapter **??**, Section **??**. This is the case, as there are no jump instructions inside the compiled expression and all the instructions inside an expression are sequential .

Our source language supports also exception handling. As we saw in the previous section, the compiler will keep track of the exception handlers by adding them in the exception handler table. We now establish that the elements in the exception handler table correspond to a statement in the source language.

Property 3.5.6 (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* $\ulcorner s, \mathcal{STMT}, e \urcorner$

Proof: 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.

In the rest of this subsection we will look at properties of the compiler which are a consequence of the upper properties. We can establish also that the compilation of a statement $\mathcal{STMT}$ may contain cycles only if $\mathcal{STMT}$ is a cycle or contains a substatement which is a cycle.

Property 3.5.7 (Cycles in the control flow graph) *The compilation* $\ulcorner s, \mathcal{STMT}, e \urcorner$ *of a* $\mathcal{STMT}$ *may contain an instruction* ins_k *which is a loop entry in the sense of definition* **??** *(i.e. there exists* ins_j *such that* $ins_j \rightarrow^l ins_k$ *) only if* $\mathcal{STMT}$ *is a loop or contains a substatement* $\mathcal{STMT}'$ *which is a loop statement and the following holds:*

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

$$\vee$$

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

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

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

Another property concerning cycles in the control flow graph is that all the instructions in a compilation $\ulcorner s, \mathcal{STMT}, e \urcorner$ of a statement $\mathcal{STMT}$ which target

13

the instruction ins_{e+1} are in the same execution relation with ins_{e+1},i.e. if ins_{e+1} is a loop entry either all are loop ends or none of them is:

Property 3.5.8 (Cycles in the control flow graph) *For every statement $\mathcal{STMT}$ its compilation $\ulcorner s, \mathcal{STMT}, e \urcorner$ is such that $(\exists k, s \le k \le e, ins_k \rightarrow^l ins_{e+1}) \iff (\forall k, s \le k \le e, ins_k \rightarrow ins_{e+1} \Rightarrow ins_k \rightarrow^l ins_{e+1})$*

Property 3.5.9 (Exception handler property for statements) *For every statement $\mathcal{STMT}$ which is not a try catch in method $\mathtt{m}$ and its substatement $\mathcal{STMT}'$, which we denote with $\mathcal{STMT}[\mathcal{STMT}']$ such that $\neg (\exists \mathcal{STMT}'', \mathcal{STMT}[\mathcal{STMT}''] \wedge \mathcal{STMT}''[\mathcal{STMT}'])$ we have that if their respective compilations are $\ulcorner s, \mathcal{STMT}, e \urcorner$ and $\ulcorner s', \mathcal{STMT}', e' \urcorner$ then the following holds:*
$$\forall \mathtt{Exc}, findExcHandler(\mathtt{Exc}, e, \mathtt{m.excHndlS}) = findExcHandler(\mathtt{Exc}, e', \mathtt{m.excHndlS})$$

Proof: The proof is by contradiction. Assume this is not true, i.e.

$\exists \mathcal{STMT}, \mathcal{STMT}',$
(1) $\quad \mathcal{STMT} \ne \mathtt{try\{\ldots\}catch\{\ldots\}} \wedge$
(2) $\quad \mathcal{STMT}[\mathcal{STMT}'] \wedge \neg (\exists \mathcal{STMT}'', \mathcal{STMT}[\mathcal{STMT}''] \wedge \mathcal{STMT}''[\mathcal{STMT}']) \wedge$
(3) $\quad \ulcorner s, \mathcal{STMT}, e \urcorner \wedge$
(4) $\quad \ulcorner s', \mathcal{STMT}', e' \urcorner \wedge$
(5) $\quad \exists \mathtt{Exc}, findExcHandler(\mathtt{Exc}, e, \mathtt{m.excHndlS}) \ne$
$\quad findExcHandler(\mathtt{Exc}, e', \mathtt{m.excHndlS})$

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 $\mathtt{m}$ $\mathtt{m.excHndlS}$ such that :

(6) $\quad eH_1 \ne eH_2$

From lemma 3.5.6 we get that :

$$
\begin{array}{c}
\exists \mathcal{STMT}_1, s_1, e_1, \ulcorner s_1, \mathcal{STMT}_1, e_1 \urcorner \\
\wedge \\
\exists \mathcal{STMT}_2, s_2, e_2, \ulcorner s_2, \mathcal{STMT}_2, e_2 \urcorner
\end{array}
\quad (7)
$$

From the initial condition *(2)* we conclude that

(8) $\quad \ulcorner s_1, \mathcal{STMT}_1, e_1 \urcorner \notin \ulcorner s, \mathcal{STMT}, e \urcorner$

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 3.5.4 we can conclude that

$$
\begin{array}{l}
\ulcorner s, \mathcal{STMT}, e \urcorner \in \ulcorner s_1, \mathcal{STMT}_1, e_1 \urcorner \in \ulcorner s_2, \mathcal{STMT}_2, e_2 \urcorner \\
\vee \\
\ulcorner s, \mathcal{STMT}, e \urcorner \in \ulcorner s_2, \mathcal{STMT}_2, e_2 \urcorner \in \ulcorner s_1, \mathcal{STMT}_1, e_1 \urcorner
\end{array}
$$

In both of the cases and of the definition of *findExcHandler* in Chapter **??**, Section **??** this means that

$findExcHandler(\mathtt{Exc}, e, \mathtt{m.excHndlS}) = findExcHandler(\mathtt{Exc}, e', \mathtt{m.excHndlS})$

which is in contradiction with *(6)*.

A similar property can be established about expressions

14

Property 3.5.10 (Exception handler property for expressions) *For every statement $\mathcal{STMT}$ which is not a try catch in method* m *and its subexpression* $\mathcal{E}^{src}$*, which we denote with* $\mathcal{STMT}[\mathcal{E}^{src}]$ *such that* $\neg (\exists \mathcal{STMT}'', \mathcal{STMT}''[\mathcal{E}^{src}] \wedge \mathcal{STMT}[\mathcal{STMT}''])$ *we have that if their respective compilations are* $\ulcorner s, \mathcal{STMT}, e \urcorner$ *and* $\ulcorner s', \mathcal{E}^{src}, e' \urcorner$ *and (2) then the following holds:*

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

The last property concerns try catch statements

Property 3.5.11 (Exception handlers and try catch statements) *For ever try catch statement* try $\{\mathcal{STMT}_1\}$ catch (ExcClass **var**)$\{\mathcal{STMT}_2\}$ *its compilation*

$$\ulcorner s, \mathcal{STMT}_1, e' \urcorner; \\ e' + 1 : \ \text{goto} \ \ e + 1; \\ \ulcorner e' + 2, \mathcal{STMT}_2, e \urcorner;$$

is such that the following holds

$$\forall \neg(\mathtt{Exc} <: \mathtt{ExcClass}) \Rightarrow \quad findExcHandler(\mathtt{Exc}, e', \mathtt{m.excHndlS}) = \\ findExcHandler(\mathtt{Exc}, e, \mathtt{m.excHndlS}) \\ \wedge findExcHandler(\mathtt{ExcClass}, e', \mathtt{m.excHndlS}) = e' + 2$$

4 Weakest precondition calculus for source programs

4.1 Source assertion language

The properties that our predicate calculus treats are from first order predicate logic. In the following, we give the formal definition of the assertion language into which the properties are encoded. Note that the language is the same to the bytecode assertion language presented earlier modulo the stack expressions `st(cntr + - ...)` and `cntr`.

Formulas 1 (Definition) *The set of formulas is defined inductively as follows*

$$
\begin{aligned}
\mathcal{F} ::=\quad & \psi(\mathcal{E}^{spec}, \mathcal{E}^{spec}) \\
& |\textbf{instances}(\mathcal{E}^{spec}) \\
& |T \\
& |\bot \\
& |\mathcal{F} \wedge \mathcal{F} \\
& |\mathcal{F} \vee \mathcal{F} \\
& |\mathcal{F} \Rightarrow \mathcal{F} \\
& |\forall x(\mathcal{F}(x)) \\
& |\exists x(\mathcal{F}(x))
\end{aligned}
$$

$$
\mathbb{P} ::= \quad == |\neq| \leq |\leq| \geq |>|<:
$$

$$
\begin{aligned}
\mathcal{E}^{spec} ::=\quad & \textbf{constInt} \\
& |\ \textbf{true} \\
& |\ \textbf{false} \\
& |\ \textbf{boundVar} \\
& |\ \mathcal{E}^{spec}\ op\ \mathcal{E}^{spec} \\
& |\ \mathcal{E}^{spec}.f \\
& |\ \textbf{var} \\
& |\ \textbf{null} \\
& |\ \textbf{this} \\
& |\ \backslash typeof(\mathcal{E}^{spec}) \\
& |\quad \backslash result
\end{aligned}
$$

4.2 Weakest Predicate Transformer for the Source Language

The weakest precondition calculates for every statement $\mathcal{STMT}$ in method $\mathtt{m}$ from our source language, for any normal postcondition $Post$ and exceptional postcondition function $\mathsf{excPost}^{src}$ ($\mathtt{Exc} \rightarrow \mathcal{STMT} \rightarrow \mathcal{F}$), the predicate Pre such that if it holds in the pre state of $\mathcal{STMT}$ and if $\mathcal{STMT}$ terminates normally then $Post$ holds in the poststate and if $\mathcal{STMT}$ terminates on exception Exc then $\mathsf{excPost}^{src}(Exc, \mathcal{STMT})$ holds. In the following, in subsection 4.2.1 we discuss how the exceptional postcondition function is defined. Subsections 4.2.2 and 4.2.3 present respectively the definition of *wp* function for expressions and for statements.

4.2.1 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 $\mathtt{Exc}$ if it terminates on exception $\mathtt{Exc}$

We introduce a function $\mathsf{excPost}^{src}$ which maps exception types to predicates

$$
\mathsf{excPost}^{src} : \ \mathtt{ETypes}\ \longrightarrow Predicate
$$

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 $\texttt{Exc}$ is thrown.

We also use function updates for $\mathsf{excPost}^{src}$ which are defined in the usual way

$$\mathsf{excPost}^{src}[\oplus\texttt{Exc'} \rightarrow P](\texttt{Exc}, exp) = \left\{ \begin{array}{ll} P & if\,\texttt{Exc} <: \texttt{Exc'} \\ \mathsf{excPost}^{src}(\texttt{Exc}, exp) & else \end{array} \right.$$

4.2.2 Expressions

The weakest precondition function for expressions has the following signature:

$$wp^{src} : \mathcal{E}^{src} \rightarrow \mathcal{F} \rightarrow (\ \texttt{Exc} \rightarrow \mathcal{F}) \rightarrow \mathbf{Method} \rightarrow \mathcal{E}^{spec} \rightarrow \mathcal{F}$$

For calculating the wp^{src} predicate of an expression $\mathcal{E}^{src}$ declared in method $\texttt{m}$, the function wp^{src} takes as arguments $\mathcal{E}^{src}$, a postcondition nPost^{src}, an exceptional postcondition function $\mathsf{excPost}^{src}$ and an $\mathcal{E}^{spec}$ and returns the formula $wp^{src}(\mathcal{E}^{src}, \psi, \mathsf{excPost}^{src}, \texttt{m})_{\mathcal{E}^{spec}}$ which is the wp precondition of $\mathcal{E}^{src}$ if the its value is represented by $\mathcal{E}^{spec}$.

In the following, we give the rules of wp^{src}.

- integer and boolean constant access
  ($const \in \{\mathbf{constInt}, \mathbf{true}, \mathbf{false}, \mathbf{constRef}\}$)

$$\mathrm{wp}^{src}(const, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \texttt{m})_{const} = \mathsf{nPost}^{src}$$

- field access expression

$$\begin{aligned} &\mathrm{wp}^{src}(\mathcal{E}_1^{src}.f, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \texttt{m})_{v.f} = \\ &\mathrm{wp}^{src}(\mathcal{E}_1^{src}, \\ &\qquad v \neq \mathbf{null} \Rightarrow \mathsf{nPost}^{src} \\ &\qquad \wedge \\ &\qquad v = \mathbf{null} \Rightarrow \mathsf{excPost}^{src}(\ \texttt{NullPointerExc}, \mathcal{E}_1^{src}) \\ &\qquad \mathsf{excPost}^{src}, \texttt{m})_v \end{aligned} \quad ,$$

- arithmetic expressions

$$\begin{aligned} &\mathrm{wp}^{src}(\mathcal{E}_1^{src}\ op\ \mathcal{E}_2^{src}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \texttt{m})_{v_1+v_2} = \\ &\mathrm{wp}^{src}(\mathcal{E}_1^{src}, \mathrm{wp}^{src}(\mathcal{E}_2^{src}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \texttt{m})_{v_2}, \mathsf{excPost}^{src}, \texttt{m})_{v_1} \end{aligned}$$

- method invokation

$$\text{wp}^{src}(\mathcal{E}^{src}.m(), \text{nPost}^{src}, \text{excPost}^{src}, \mathbf{m})_v =$$
$$\text{wp}^{src}(\mathcal{E}^{src},$$

$$\left\{ \begin{array}{l} v' \neq \ \mathbf{null} \Rightarrow \\ \quad m.\text{Pre}^{src}[\mathbf{this} \leftarrow v'] \\ \quad \wedge \\ \quad \forall \mathbf{boundVar}, \ \forall \ m \in m.\text{modif}^{src} \\ \quad \left\{ \begin{array}{l} \text{\textbackslash typeof}(\mathbf{boundVar}) <: m.\text{retType} \wedge \\ m.\text{nPost}^{src} \left[\begin{array}{l} \text{\textbackslash result} \ \leftarrow \mathbf{boundVar}] \\ [\mathbf{this} \leftarrow v'] \end{array} \right. \\ \quad \Rightarrow \text{nPost}^{src}[v \leftarrow \mathbf{boundVar}] \end{array} \right. \\ \quad \wedge \\ \quad \forall \mathtt{E} \in m.\text{exceptions}^{src}, \\ \quad \forall \ m \in m.\text{modif}^{src} \\ \quad \quad m.\text{excPostSpec}^{src}(\mathtt{E}) \Rightarrow \text{excPost}^{src}(\mathtt{E}) \\ \mathcal{E}_1^{src} = \ \mathbf{null} \Rightarrow \text{excPost}^{src}(\ \mathtt{NullPntrExc}) \end{array} \right. ,$$

$$\text{excPost}^{src}, \mathbf{m})_{v'}$$

- Cast expression

$$\text{wp}^{src}((\ \mathtt{Class}\ )\ \mathcal{E}^{src}, \text{nPost}^{src}, \text{excPost}^{src}, \mathbf{m})_v =$$
$$\text{wp}^{src}(\mathcal{E}^{src},$$

$$\left. \begin{array}{l} \text{\textbackslash typeof}(v) <: \ \mathtt{Class}\ \Rightarrow \\ \quad \quad \text{nPost}^{src} \\ \wedge \\ \neg\ \text{\textbackslash typeof}(v) <: \ \mathtt{Class}\ \Rightarrow \\ \quad \quad \text{excPost}^{src}(\ \mathtt{CastExc}) \end{array} \right. ,$$

$$\text{excPost}^{src}, \mathbf{m})_v$$

- Null expression

$$\text{wp}^{src}(\mathbf{null}, \text{nPost}^{src}, \text{excPost}^{src}, \mathbf{m})_{\mathbf{null}} = \text{nPost}^{src}$$

- this

$$\text{wp}^{src}(\mathbf{this}, \text{nPost}^{src}, \text{excPost}^{src}, \mathbf{m})_{\mathbf{this}} = \text{nPost}^{src}$$

- instance creation

$$\mathrm{wp}^{src}(\mathbf{new}\ \texttt{Class}\ (\mathcal{E}^{src}), \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_v =$$
$$\mathrm{wp}^{src}(\mathcal{E}^{src},$$

$$\left\{
\begin{array}{l}
\forall \mathbf{boundVar}, \\
\quad not\ \mathbf{instances}(\mathbf{boundVar}) \wedge \\
\quad \mathbf{boundVar} \neq \mathbf{null} \Rightarrow \\
\qquad \mathsf{constr}(\ \texttt{Class}\ ).\mathsf{Pre}^{src}\ [arg \leftarrow v'] \\
\qquad \wedge \\
\qquad \forall\ m \in \mathsf{constr}(Class).\mathsf{modif}^{src}, \\
\qquad\qquad\qquad\qquad\qquad [\mathbf{this} \leftarrow \mathbf{boundVar}] \\
\qquad\quad \mathsf{constr}(Class).\mathsf{nPost}^{src}\ [arg \leftarrow v'] \\
\qquad\qquad\qquad\qquad\qquad [\backslash\texttt{typeof}\,(\mathbf{boundVar}) \leftarrow Class] \\
\qquad\quad \Rightarrow \mathsf{nPost}^{src}[v \leftarrow \mathbf{boundVar}] \qquad \wedge \\
\qquad \forall \texttt{Exc} \in \mathsf{constr}(Class).\mathsf{exceptions}^{src}, \\
\qquad \forall\ m \in \mathsf{constr}(Class).\mathsf{modif}^{src}, \\
\qquad\quad \mathsf{constr}(Class).\mathsf{excPostSpec}^{src}(\texttt{Exc}) \Rightarrow \mathsf{excPost}^{src}(\texttt{Exc})
\end{array}
\right. ,$$

$$\mathsf{excPost}^{src}, \mathtt{m})_{v'}$$

Let us see the relational expressions supported in the source programming language

- Instanceof expression

$$\mathrm{wp}^{src}(\mathcal{E}^{src}\ instanceof\ \texttt{Class}\ , \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_{\backslash\texttt{typeof}\,(v)\ <:\ \texttt{Class}\ \wedge v \neq \mathbf{null}} =$$
$$\mathrm{wp}^{src}(\mathcal{E}^{src}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_v$$

- Binary relation over expressions

$$\mathrm{wp}^{src}(\mathcal{E}_1^{src}\ \mathcal{R}\ \mathcal{E}_2^{src}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_{v_1 \mathcal{R} v_2} =$$
$$\mathrm{wp}^{src}(\mathcal{E}_1^{src}, \mathrm{wp}^{src}(\mathcal{E}_2^{src}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})_{v_2}, \mathsf{excPost}^{src}, \mathtt{m})_{v_1}$$

4.2.3 Statements

In the following, we present the rules of the predicate transformer for control statements.

- integer and boolean constant access

$$\mathrm{wp}^{src}(\mathcal{STMT}_1; \mathcal{STMT}_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =$$

$$\mathrm{wp}^{src}(\mathcal{STMT}_1, \mathrm{wp}^{src}(\mathcal{STMT}_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}), \mathsf{excPost}^{src}, \mathtt{m})$$

- assignment

  - local variable assignemnt

$$\mathrm{wp}^{src}(\mathcal{E}_1^{src} = \mathcal{E}_2^{src}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =$$

$$\mathrm{wp}^{src}(\mathcal{E}_2^{src},$$
$$\quad \mathsf{nPost}^{src}[\mathcal{E}_1^{src} \leftarrow v_2],$$
$$\quad \mathsf{excPost}^{src}, \mathtt{m})_{v_2}$$

– instance field assignemnt

$$\mathrm{wp}^{src}(\mathcal{E}_1^{src}.f = \mathcal{E}_2^{src}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =$$

$$
\begin{aligned}
\mathrm{wp}^{src}(\mathcal{E}_1^{src}, \\
\quad \mathrm{wp}^{src}(\mathcal{E}_2^{src}, \\
\qquad\quad v_1 \neq \mathbf{null} \Rightarrow \\
\qquad\qquad\quad \mathsf{nPost}^{src}[f \leftarrow f[\oplus v_1 \rightarrow v_2]] \\
\qquad\quad \wedge \\
\qquad\quad v_1 = \mathbf{null} \Rightarrow \\
\qquad\qquad\quad \mathsf{excPost}^{src}(\mathtt{NullPointerExc}) \\
\qquad\quad \mathsf{excPost}^{src}, \mathtt{m})_{v_2}, \\
\quad \mathsf{excPost}^{src}, \mathtt{m})v_1
\end{aligned},
$$

- if statement

$$\mathrm{wp}^{src}\left(\begin{array}{l} \mathtt{if}\ (\mathcal{E}^{\mathcal{R}}) \\ \mathtt{then}\{\mathcal{STMT}_1\} \\ \mathtt{else}\ \{\mathcal{STMT}_2\} \end{array}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m} \right) =$$

$$
\begin{aligned}
\mathrm{wp}^{src}(\mathcal{E}^{\mathcal{R}}, \\
\quad v = \mathbf{true} \Rightarrow \mathrm{wp}^{src}(\mathcal{STMT}_1, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) \\
\quad \wedge \\
\quad v = \mathbf{false} \Rightarrow \mathrm{wp}^{src}(\mathcal{STMT}_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) \\
\quad \mathsf{excPost}^{src}, \mathtt{m})_v
\end{aligned},
$$

- throw exceptions

$$\mathrm{wp}^{src}(\mathtt{throw}\ \mathcal{E}^{src}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =$$

$$
\begin{aligned}
\mathrm{wp}^{src}(\mathcal{E}^{src}, \\
\quad v = \mathbf{null} \Rightarrow \mathsf{excPost}^{src}(\mathtt{NullPointerExc}) \\
\quad \wedge \\
\quad v \neq \mathbf{null} \Rightarrow \\
\qquad \forall \mathtt{Exc}\ , \\
\qquad \backslash \mathtt{typeof}(\mathcal{E}^{src}) <: \mathtt{Exc}\ \Rightarrow \\
\qquad\quad \mathtt{m.excPost}^{src}(\mathtt{Exc}\ ) \\
\quad \mathsf{excPost}^{src}, \mathtt{m})_v
\end{aligned},
$$

- try catch statement

$$\mathrm{wp}^{src}(\mathtt{try}\ \{\mathcal{STMT}_1\}\ \mathtt{catch}(\mathtt{Exc}\ c)\ \{\mathcal{STMT}_2\}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =$$

$$
\begin{aligned}
\mathrm{wp}^{src}(\mathcal{STMT}_1, \\
\quad \mathsf{nPost}^{src}, \\
\quad \mathsf{excPost}^{src} \oplus [\mathtt{Exc} \longrightarrow \mathrm{wp}^{src}(\mathcal{STMT}_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m})], \mathtt{m})
\end{aligned}
$$

- try finally

$$\mathrm{wp}^{src}(\texttt{try } \{\mathcal{STMT}_1\} \texttt{ finally } \{\mathcal{STMT}_2\}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =$$

$$\mathrm{wp}^{src}(\mathcal{STMT}_1,$$
$$\mathrm{wp}^{src}(\mathcal{STMT}_2, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}),$$
$$\mathsf{excPost}^{src} \oplus [\texttt{Exception} \longrightarrow \mathrm{wp}^{src}(\mathcal{STMT}_2, \mathsf{excPost}^{src}(\texttt{Exception}), \mathsf{excPost}^{src}, \mathtt{m})], \mathtt{m})$$

where *exc* is the exception object thrown by $\mathcal{STMT}_1$.

- loop statement

$$\mathrm{wp}^{src}(\texttt{while } (\mathcal{E}^{\mathcal{R}}) \texttt{ [INV, modif] } \{\mathcal{STMT}\}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =$$

INV $\wedge$
$\forall\, m, m \in \texttt{modif},$
    INV $\Rightarrow$
        $\mathrm{wp}^{src}(\mathcal{E}^{\mathcal{R}},$
            $v = \mathbf{true} \Rightarrow \mathrm{wp}^{src}(\mathcal{STMT}, \texttt{INV}, \mathsf{excPost}^{src}, \mathtt{m})$
            $v = \mathbf{false} \Rightarrow \mathsf{nPost}^{src}$,
          $\mathsf{excPost}^{src}, \mathtt{m})_v$

- return statement

$$\mathrm{wp}^{src}(\texttt{return } \mathcal{E}^{src}, \mathsf{nPost}^{src}, \mathsf{excPost}^{src}, \mathtt{m}) =$$
$$\mathrm{wp}^{src}(\mathcal{E}^{src}, \mathsf{nPost}^{src}[\quad \backslash\mathrm{result} \ \leftarrow v], \mathsf{excPost}^{src}, \mathtt{m})_v$$

where \result is a specification variable that can be met in the post-condition and denotes to the value returned of a non void method

5 Weakest precondition calculus for bytecode programs

5.1 Weakest predicate transformer over compiled statements and expressions

In the following, we introduce a new formulation of the *wp* function which will be based on the compiler from source to bytecode language. The 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 **??**, section **??**. We will argue under what conditions both 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 , jsr , ret) which takes explicitly the postcondition and the exceptional postcondition function upon which the precondition will be calculated. Its signature is the following:

$$\mathrm{wp}^{bc} : \ \mathrm{I} \backslash \{ \text{ goto , if} \} \to \mathcal{F}^{bc} \to (\ \texttt{ExcType} \to \mathcal{F}^{bc}) \to \mathcal{F}^{bc}$$

For instance, the *wp* definition for getfield is :

$$wp^{bc}(\text{ getfield } f, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) =$$
$$\mathtt{st(cntr\ )} \neq \mathbf{null} \Rightarrow \psi[\mathtt{st(cntr\ )} \leftarrow f(\mathtt{st(cntr\ )})]\wedge$$
$$\mathtt{st(cntr\ )} = \mathbf{null} \Rightarrow \mathsf{excPost}^{src2bc}(\mathtt{NullPntrExc})$$

Note that this differs from the definition of the *wp* given in Chapter **??**, section **??** where the postcondition is a function of the successor of the current instruction. We do not give the rest of the rules because they the same as the rules presented in **??** except for the fact that the local postconditions are given explicitely.

We also define the weakest predicate transformer function for a block of instruction that always execute sequentially as follows:

Definition 5.1.1 (*wp* for a block of instructions)

$$wp^{bc}_{seq}(ins_1; ...; ins_k, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) =$$
$$wp^{bc}_{seq}(ins_1; ...; ins_{k-1}, wp^{bc}(ins_k, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}), \mathsf{excPost}^{src2bc}, \mathtt{m})$$

We turn now to the rules for compiled expressions. Note that from Property 3.5.3 it follows that the compilation of any expression is a sequence of instructions 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 5.1.2 (*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* $\ulcorner\mathcal{E}\urcorner$ *is* $wp^{bc}_{seq}(\ulcorner\mathcal{E}\urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})$

For instance, the rule of the *wp* for the compilation of access field expression $\mathcal{E}^{src}.f$ where its compilation is

$$\ulcorner s, \mathcal{E}^{src}_1; , e-1 \urcorner$$
$$e \quad \text{getfield } f$$

produce the following formula

$$wp^{bc}_{seq}(\begin{array}{l} \ulcorner s, \mathcal{E}^{src}, e-1 \urcorner; \\ e \quad \text{getfield } f \end{array} , \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})$$

This is equivalent to :

$$wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{src}, e-1 \urcorner, wp^{bc}(\text{ getfield } f, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}), \mathsf{excPost}^{src2bc}, \mathtt{m})$$

The function which calculates the *wp* predicate of a compiled statement is called wp^{bc}_{stmt} and has the following signature :

$$wp^{bc}_{stmt} : Set(\text{ I}) \rightarrow \mathcal{F}^{bc} \rightarrow (\mathtt{Exc} \rightarrow \mathcal{F}^{bc}) \rightarrow \mathcal{F}^{bc}$$

The definition of wp^{bc}_{stmt} uses the compiler function defined in Section **??**

- sequential statement compilation $\ulcorner s, \mathcal{STMT}_1; \mathcal{STMT}_2, e \urcorner$ which by definition is

$$wp^{bc}_{stmt}(\ulcorner s, \mathcal{STMT}_1; \mathcal{STMT}_2, e \urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) =^{def}$$
$$wp^{bc}_{stmt}(\ulcorner s, \mathcal{STMT}_1, e' \urcorner;,$$
$$\qquad wp^{bc}_{stmt}(\ulcorner e' + 1, \mathcal{STMT}_2, s \urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}),$$
$$\qquad \mathsf{excPost}^{src2bc}, \mathtt{m})$$

- if statement compilation $\ulcorner s, \mathtt{if}\ (\mathcal{E}^{\mathcal{R}})\ \mathtt{then}\ \{\mathcal{STMT}_1\}\ \mathtt{else}\ \{\mathcal{STMT}_2\}, e \urcorner$

$$wp^{bc}_{stmt}(\ulcorner s, \begin{array}{l} \mathtt{if}\ (\mathcal{E}^{\mathcal{R}}) \\ \mathtt{then}\ \{\mathcal{STMT}_1\} \\ \mathtt{else}\ \{\mathcal{STMT}_2\} \end{array}, e \urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) =^{def}$$

$$wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{\mathcal{R}}, e' \urcorner;$$
,
$$\begin{array}{l} \mathcal{R}(\mathtt{st(cntr\ )}, \mathtt{st(cntr\ -\ 1\ )}) \Rightarrow \\ \qquad wp^{bc}_{stmt}(\ulcorner e'' + 2, \mathcal{STMT}_1, e \urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})[t \leftarrow t - 2] \\ \wedge \\ \neg\mathcal{R}(\mathtt{st(cntr\ )}, \mathtt{st(cntr\ -\ 1\ )}) \Rightarrow \\ \qquad wp^{bc}_{stmt}(\ulcorner e' + 2, \mathcal{STMT}_2, e'' \urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})[t \leftarrow t - 2] \\ \mathsf{excPost}^{src2bc}, \mathtt{m}) \end{array},$$

- assignment expression. We will look only at the case for compiled field assignment expressions $\ulcorner s, \mathcal{E}^{src}_1.f = \mathcal{E}^{src}_2, e \urcorner$.

$$wp^{bc}_{stmt}(\ulcorner s, \mathcal{E}^{src}_1.f = \mathcal{E}^{src}_2, e \urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) =^{def}$$

$$wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{src}_1, e' \urcorner,$$

$$\qquad wp^{bc}_{seq}(\begin{array}{l} \ulcorner e' + 1, \mathcal{E}^{src}_2, e - 1 \urcorner; \\ e \quad \mathtt{putfield}\ f \end{array}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}),$$

$$\qquad \mathsf{excPost}^{src2bc}, \mathtt{m})$$

- try catch statement compilation

$$wp^{bc}_{stmt}(\ulcorner s, \begin{array}{l} \mathtt{try}\ \{\mathcal{STMT}_1\} \\ \mathtt{catch}\ (\mathtt{ExcClass}\ \mathbf{var})\{\mathcal{STMT}_2\} \end{array}, e \urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) =^{def}$$

$$wp^{bc}_{stmt}(\begin{array}{l} \ulcorner s, \mathcal{STMT}_1, e' \urcorner; \\ e' + 1 : \ \mathtt{goto}\ \ e + 1; \end{array},$$

$$\qquad \psi,$$

$$\qquad \mathsf{excPost}^{src2bc}[\oplus\mathtt{ExcClass} \rightarrow wp^{bc}_{stmt}(\ulcorner e' + 2, \mathcal{STMT}_2, e \urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})], \mathtt{m})$$

- try finally statement compilation $\ulcorner s, \mathtt{try}\ \{\mathcal{STMT}_1\}\ \mathtt{finally}\ \{\mathcal{STMT}_2\}, e \urcorner$

$$wp^{bc}_{stmt}(\ulcorner s,\ \begin{array}{l}\mathtt{try}\ \{\mathcal{STMT}_1\}\\ \mathtt{finally}\ \{\mathcal{STMT}_2\}\end{array}\ , e \urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) =^{def}$$

$$wp^{bc}_{stmt}(\ulcorner s, \mathcal{STMT}_1, e' \urcorner$$
,

$$wp^{bc}_{stmt}(\ulcorner e' + 1, \mathcal{STMT}_2, e'' \urcorner; , \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}),$$

$$\mathsf{excPost}^{src2bc}[\oplus\mathtt{ExcClass} \rightarrow wp^{bc}_{stmt}(\ \begin{array}{l} e'' + 2: \ \mathtt{store}\ l;\\ \ulcorner e'' + 3, \mathcal{STMT}_2, e - 2 \urcorner;\\ e - 1: \ \mathtt{load}\ l;\\ e: \ \mathtt{athrow}\end{array}\ , \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})], \mathtt{m})$$

- throw exception compilation $\ulcorner s, \mathtt{throw}\ \mathcal{E}^{src}, e \urcorner$

$$wp^{bc}_{stmt}(\ulcorner s, \mathtt{throw}\ \mathcal{E}^{src}, e \urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) =^{def}$$
$$wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{src}, e - 1 \urcorner, wp^{bc}(e/\ \mathtt{athrow}, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}), \mathsf{excPost}^{src2bc}, \mathtt{m})$$

- loop statement

$$wp^{bc}_{stmt}(\ulcorner s,\ \begin{array}{l}\mathtt{while}\ (\mathcal{E}^{\mathcal{R}})\\ [\mathtt{INV},\mathtt{modif}]\\ \{\mathcal{STMT}\}\end{array}\ , e \urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) =^{def}$$

$$\begin{array}{l}\mathtt{INV}\\ \wedge\\ \forall mod \in \mathtt{modif},\\ \mathtt{INV} \Rightarrow\\ wp^{bc}_{seq}(\ulcorner e' + 1, \mathcal{E}^{\mathcal{R}}, e - 1 \urcorner,\\ \qquad \mathcal{R}(\mathtt{st(cntr)}, \mathtt{st(cntr - 1)}) \Rightarrow\\ \qquad\qquad wp^{bc}_{stmt}(\ulcorner s + 1, \mathcal{STMT}, e' \urcorner, \mathtt{INV}, \mathsf{excPost}^{src2bc}, \mathtt{m})\\ \qquad \wedge\\ \qquad \neg\mathcal{R}(\mathtt{st(cntr)}, \mathtt{st(cntr - 1)}) \Rightarrow \psi\\ \mathsf{excPost}^{src2bc}, \mathtt{m})\end{array}$$
,

5.2 Properties of the wp functions

The previous subsection introduced a new formulation of the wp function for bytecode which takes into account 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 wp**??**. The following statements establish the relation between the two versions of the wp calculus.

Lemma 5.2.1 (Equivalence of the formulations for single instructions)
For all instructions $i_j \in$ I\\{ goto , if, jsr, ret} and i_k which belong to method
m, *formula ψ,and function* $\mathsf{excPost}^{src2bc}:$ ExcType $\rightarrow \mathcal{F}^{bc}$ *such that*

- $i_j \rightarrow i_k$

- $\psi = inter(i_j, i_k)$

- $\forall \texttt{Exc}, \textsf{excPost}^{src2bc}(\texttt{Exc}) = \texttt{m}.\textsf{excPost}(\texttt{Exc}, j)$

the following holds
$wp^{bc}(i_j, \psi, \textsf{excPost}^{src2bc}, \texttt{m}) = wp(i_j , \texttt{m})$

The proof is done by case analysis on the instruction i_j In the following, we scatch the case for a getfield instruction *Proof:*

{ *by hypothesis* }
$i_j =$ getfield f
{ *by definition of the wp function* }
(1) $wp^{bc}($ getfield $f, \psi, \textsf{excPost}^{src2bc}, \texttt{m}) =$
$\texttt{st(cntr)} \neq \textbf{null} \Rightarrow \psi[\texttt{st(cntr)} \leftarrow f(\texttt{st(cntr)})]$
$\wedge$
$\texttt{st(cntr)} = \textbf{null} \Rightarrow \textsf{excPost}^{src2bc}(\texttt{NullPntrExc})$

{ *by definition of the* wp *function* }
(2) $wp($ getfield $f , \texttt{m}) =$
$\texttt{st(cntr)} \neq \textbf{null} \Rightarrow inter(i_j, i_k) [\texttt{st(cntr)} \leftarrow f(\texttt{st(cntr)})]$
$\wedge$
$\texttt{st(cntr)} = \textbf{null} \Rightarrow \textsf{excPost}(\texttt{NullPntrExc}, j)$

{ *from the initial hypothesis, (1) and (2) the lemma holds in that case* }

Lemma 5.2.2 (*wp* **for a block of instructions)** *For every block of instructions* $ins_1; \ldots; ins_j$ *and instruction* ins_k *in method* $\texttt{m}$*, formula* ψ *and function* $\textsf{excPost}^{src2bc} : \texttt{ExcType} \rightarrow \mathcal{F}^{bc}$ *such that*

- $ins_j \rightarrow ins_k$

- $\psi = inter(ins_j, ins_{j+1})$

- $\forall \texttt{Exc}, \forall i, 1 \leq i \leq j, \textsf{excPost}^{src2bc}(\texttt{Exc}) = \texttt{m}.\textsf{excPost}(\texttt{Exc}, i)$

then the following holds

$$wp^{bc}_{seq}(ins_1; \ldots; ins_j, \psi, \textsf{excPost}^{src2bc}, \texttt{m}) = wp(ins_1 , \texttt{m})$$

The proof is done by induction on the length of the sequence of instructions. *Proof:*

25

(1) $wp^{bc}_{seq}(ins_1; \ldots; ins_j, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) =$
{ *by Def. 5.1.1 for wp of block of instructions* }
$wp^{bc}_{seq}(ins_1; \ldots; ins_{j-1}, wp^{bc}(ins_j, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}), \mathsf{excPost}^{src2bc}, \mathtt{m})$
{ *by initial hypothesis we can apply lemma 5.2.1 from which it follows* }
$wp^{bc}(ins_j, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = wp(\,ins_j, \mathtt{m})$
{*by definition 3.5.1 , ins_j is not a loop entry*
 and from Def. **??** }
(2) $inter(j, j+1) = wp(\,ins_j, \mathtt{m})$
{*apply the induction hypothesis over $ins_1; \ldots; ins_{j-1}$,*
 (2) and the initial hypothesis for exception handlers }
(3) $wp^{bc}_{seq}(ins_1; \ldots; ins_{j-1}, wp^{bc}(ins_j, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}), \mathsf{excPost}^{src2bc}, \mathtt{m}) =$
$wp(\,ins_1, \mathtt{m})$
{ *from (1) and (3) the proposition holds*}

Lemma 5.2.3 (*wp* **for compiled expressions**) *For every compiled expression $\ulcorner s, \mathcal{E}^{src}, e \urcorner$ in method $\mathtt{m}$ formula ψ and function $\mathsf{excPost}^{src2bc} : \mathtt{ExcType} \rightarrow \mathcal{F}^{bc}$ such that*

- $\psi = inter(e, e+1)$

- $\forall \mathtt{Exc}, \forall 1 \le i, k \le j, \mathsf{excPost}^{src2bc}(\mathtt{Exc}) = \mathtt{m.excPost}(\mathtt{Exc}, i)$

then the following holds:

$$wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{src}, e \urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = wp(ins_s\,, \mathtt{m})$$

Proof: From Property 3.5.5 of the compiler it follows that for every expression $\mathcal{E}^{src}$, start label s and end label e, the resulting compilation $\ulcorner s, \mathcal{E}^{src}, e \urcorner$ is a block of instructions. We can apply the previous lemma 5.2.2 and we get the result.

Lemma 5.2.4 *For every compiled statement $\ulcorner s, \mathcal{STMT}, e \urcorner$ in method $\mathtt{m}$, formula ψ and function $\mathsf{excPost}^{src2bc} : \mathtt{ExcType} \rightarrow \mathcal{F}^{bc}$ such that*

- $\psi = inter(e, e+1)$

- $\forall \mathtt{Exc}, \mathsf{excPost}^{src2bc}(\mathtt{Exc}) = \mathtt{m.excPost}(\mathtt{Exc}, e)$

then the following holds: if $\psi = inter(e, e+1)$ then the following holds

$$wp^{bc}_{stmt}(\ulcorner s, \mathcal{STMT}, e \urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = wp(ins_s\,, \mathtt{m})$$

Proof : the proof is by induction on the compilation of a statement. We scatch here the proof of few cases

- if statement

$$wp^{bc}_{stmt}(\ulcorner s, \ \begin{array}{l}\texttt{if } (\mathcal{E}^{\mathcal{R}})\\ \texttt{then } \{\mathcal{STMT}_1\}\\ \texttt{else } \{\mathcal{STMT}_2\}\end{array}, e\urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) =^{def}$$

$$wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{\mathcal{R}}, e'\urcorner;$$

$$\left(\begin{array}{l}
\mathcal{R}(\mathtt{st(cntr\ )}, \mathtt{st(cntr\ -\ 1\ )}) \Rightarrow \\
\qquad wp^{bc}_{stmt}(\ulcorner e'' + 2, \mathcal{STMT}_1, e\urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})[t \leftarrow t - 2] \\
\wedge \\
\neg\mathcal{R}(\mathtt{st(cntr\ )}, \mathtt{st(cntr\ -\ 1\ )}) \Rightarrow \\
\qquad wp^{bc}_{stmt}(\ulcorner e' + 2, \mathcal{STMT}_2, e''\urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})[t \leftarrow t - 2] \\
\mathsf{excPost}^{src2bc}, \mathtt{m})
\end{array}\right),$$

{ by initial hypothesis, we have }
(1.1) $\psi = inter(e, e + 1)$
{ by the compiler definition $ins_{e''+1} = $ goto $e + 1$; }
(1.2) $ins_{e''+1} \rightarrow ins_{e+1}$
{ from Property 3.5.8 which states that every instruction inside
the compilation of $\ulcorner s, \mathcal{STMT}, e\urcorner$,
is in the same execution relation with ins_{e+1},

Def. ?? for inter, (1.1) and (1.2) we conclude that }

(1.3) $inter(e'' + 1, e + 1) = inter(e, e + 1) = \psi$

{*From property 3.5.7 the edge between e'' and $e'' + 1$*
is not a loop edge and from Def. ?? }

(1.4) $inter(e'', e'' + 1) = wp(\ e'' + 1, m)$

{ *From the definition of the wp for* goto *instructions and (1.4)* }

(1.5) $wp(\ e'' + 1, m) = inter(e'' + 1, e + 1)$

{ *From (1.4), (1.3) and (1.5)*}

(1.6) $inter(e'', e'' + 1) = inter(e, e + 1) = \psi$

{ *From 3.5.9* }

(1.7) $\mathtt{m.excPost}(\mathtt{Exc}, e) = \mathtt{m.excPost}(\mathtt{Exc}, e'')$

{ *apply the induction hypothesis over (1.6) and (1.7)* }

(1.8) $wp^{bc}_{stmt}(\ulcorner e' + 2, \mathcal{STMT}_2, e''\urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = wp(\ e' + 2, m)$

{ *apply the induction hypothesis over the initial hypothesis* }

(1.9) $wp^{bc}_{stmt}(\ulcorner e'' + 2, \mathcal{STMT}_1, e\urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = wp(\ e'' + 2, m)$

{ *by definition of* if *function* }

(1.10) $wp(\ e' + 1: \ \mathtt{if}\ e'' + 2; , m) =$

$\mathcal{R}(\mathtt{st(cntr\ )}, \mathtt{st(cntr\ -\ 1\ )}) \Rightarrow wp(\ e'' + 2, m)$

$\wedge$

$\neg \mathcal{R}(\mathtt{st(cntr\ )}, \mathtt{st(cntr\ -\ 1\ )}) \Rightarrow wp(\ e' + 2, m)$

{ *from (1.8), (1.9) and (1.10)*}

(1.11) $wp(\ e' + 1: \ \mathtt{if}\ e'' + 2; , m) =$

$\mathcal{R}(\mathtt{st(cntr\ )}, \mathtt{st(cntr\ -\ 1\ )}) \Rightarrow$

$\qquad wp^{bc}_{stmt}(\ulcorner e'' + 2, \mathcal{STMT}_1, e\urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})$

$\wedge$

$\neg \mathcal{R}(\mathtt{st(cntr\ )}, \mathtt{st(cntr\ -\ 1\ )}) \Rightarrow$

$\qquad wp^{bc}_{stmt}(\ulcorner e' + 2, \mathcal{STMT}_2, e''\urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})$

{ *From Property 3.5.8* }

(1.12) $inter(e', e' + 1) = wp(\ e' + 1: \ \mathtt{if}\ e'' + 2; , m)$

{ *From the initial hypothesis* }

(1.13) $\mathtt{m.excPost}(\mathtt{Exc}, e) = \mathsf{excPost}^{src2bc}(\mathtt{Exc})$

{ *From Property 3.5.10 and (1.13)*}

(1.14) $\forall \mathtt{Exc}, \forall s \leq i \leq e', \mathsf{excPost}^{src2bc}(\mathtt{Exc}) =$

$\mathtt{m.excPost}(\mathtt{Exc}, i)$

{ *apply Lemma 5.2.3 over hypothesis (1.11), (1.12), (1.14)* }

$wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{\mathcal{R}}, e'\urcorner;$

,

$\qquad \mathcal{R}(\mathtt{st(cntr\ )}, \mathtt{st(cntr\ -\ 1\ )}) \Rightarrow$

$\qquad\qquad wp^{bc}_{stmt}(\ulcorner e'' + 2, \mathcal{STMT}_1, e\urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})[t \leftarrow t - 2]$

$\qquad \wedge$ $\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad$,

$\qquad \neg \mathcal{R}(\mathtt{st(cntr\ )}, \mathtt{st(cntr\ -\ 1\ )}) \Rightarrow$

$\qquad\qquad wp^{bc}_{stmt}(\ulcorner e' + 2, \mathcal{STMT}_2, e''\urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})[t \leftarrow t - 2]$

$\qquad \mathsf{excPost}^{src2bc}, \mathtt{m}) = wp(\ s, m)$

{ *and this case holds*}

- try catch statement

(1) $wp^{bc}_{stmt}(\ulcorner s, \begin{array}{l} \texttt{try } \{\mathcal{STMT}_1\} \\ \texttt{catch (ExcClass var)}\{\mathcal{STMT}_2\} \end{array}, e\urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) =^{def}$

$\{\ \textit{Def. of } wp^{bc}_{stmt} \textit{ in the previous Section 5.1 }\}$

$wp^{bc}_{stmt}(\ulcorner s, \mathcal{STMT}_1, e'\urcorner; ,$

$\qquad \psi,$

$\qquad \mathsf{excPost}^{src2bc}[\oplus \texttt{ExcClass} \rightarrow wp^{bc}_{stmt}(\ulcorner e'+2, \mathcal{STMT}_2, e\urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})], \mathtt{m})$

$\{\ \textit{apply the induction hypothesis over } \mathcal{STMT}_2 \textit{ and the initial hypothesis}\}$

(2) $wp^{bc}_{stmt}(\ulcorner e'+2, \mathcal{STMT}_2, e\urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m}) = wp(\ e'+2, \mathtt{m})$

$\{\ \textit{from the definition of } \mathsf{excPost} \textit{ and property 3.5.11}\}$

(3) $\forall \texttt{Exc}, \mathsf{excPost}^{src2bc}[\oplus \texttt{ExcClass} \rightarrow wp(\ e'+2, \mathtt{m})] =$

$\qquad \mathtt{m}.\mathsf{excPost}(\texttt{Exc}, e')$

$\{\ \textit{we can also conclude from Prop. 3.5.8 and 3.5.7 }\}$

(4) $inter(e', e+1) = inter(e+1, e+1) = inter(e, e+1)$

$\{\ \textit{apply induction hypothesis over (1), (3) and (4) }\}$

$wp^{bc}_{stmt}(\ulcorner s, \mathcal{STMT}_1, e'\urcorner; ,$

$\qquad \psi,$

$\qquad \mathsf{excPost}^{src2bc}[\oplus \texttt{ExcClass} \rightarrow wp^{bc}_{stmt}(\ulcorner e'+2, \mathcal{STMT}_2, e\urcorner, \psi, \mathsf{excPost}^{src2bc}, \mathtt{m})], \mathtt{m}) =$

$wp(\ s, \mathtt{m})$

[section] [section] [section]

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 6.1 (Normal form of wp^{src}) *for any source expression $\mathcal{E}^{src}$ normal form of wp^{src} is such that*

either there exists $Q, R : \mathcal{F}$
$wp^{src}(\mathcal{E}^{src}, \psi, \mathsf{excPost}, \mathtt{m})_v \equiv$
$Q \Rightarrow \psi$
$\wedge$
R

or exists $Q, R : \mathcal{F}, v_1 \ldots v_k : \mathcal{E}^{src}$

$wp^{src}(\mathcal{E}^{src}, \psi, \mathsf{excPost}, \mathtt{m})_v \equiv$

$\forall \mathbf{boundVar}_1, \ldots \mathbf{boundVar}_k, Q \Rightarrow \psi \begin{array}{l}[v_1 \leftarrow \mathbf{boundVar}_1] \\ \ldots \\ [v_k \leftarrow \mathbf{boundVar}_k]\end{array} \wedge$

R

This follows from the definition of wp^{src} in Section 4.2.2. 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 $\ulcorner s, \mathcal{E}^{src}, e \urcorner$. 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 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 $\phi : \mathcal{F}^{bc}$ such that ϕ may only contain stack expressions of the form* st(cntr - k) $, k \geq 0$, *the following holds*
There exist $Q, R : \mathcal{F}$ such that

$$wp^{src}(\mathcal{E}^{src}, \psi, \mathsf{excPost}, \mathtt{m})_v \;\equiv$$
$$Q \Rightarrow \psi$$
$$\wedge$$
$$R$$

$$\Rightarrow$$

$$wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{src}, e \urcorner, \phi, \mathsf{excPost}, \mathtt{m}) \;\equiv$$
$$Q \Rightarrow \phi \begin{bmatrix} \mathtt{cntr} \;\leftarrow\; \mathtt{cntr} \;+1 \\ [\mathtt{st(cntr +1)} \;\leftarrow\; v] \end{bmatrix}$$

$$\wedge$$
$$R$$

We proceed with several cases of the proof, which is done by induction over the structure of the formula

Proof :

1. $\mathcal{E}^{src} = const, const \in$ **constInt**, **true**, **false**

{ *source case* }

(1)$\mathrm{wp}^{src}(const, \psi, \mathsf{excPost}, \mathtt{m})_{const}$

{ *following the definition of the wp function for source expressions in subsection 4.2.2* }

$\equiv \psi$

{ *bytecode case* }

(2)$wp_{seq}^{bc}(\ulcorner s, const, s \urcorner, \phi, \mathsf{excPost}, \mathtt{m})$

{ *following the definition of the compiler function in subsection 3.3* }

$\equiv wp^{bc}(s \; \text{push} const, \phi, \ulcorner \mathsf{excPost} \urcorner, \mathtt{m})$

{ *following the definition of the wp function for bytecode in subsection 4.2.2* }

$\equiv \phi \begin{bmatrix} \mathtt{cntr} \; \leftarrow \mathtt{cntr} \; + 1 \\ \mathtt{st(} \; \mathtt{cntr} \; \mathtt{+1)} \; \leftarrow const \end{bmatrix}$

{ *from (1) and (2) and $Q, R = T$ this case holds* }

2. $\mathcal{E}^{src} = \mathcal{E}^{src}.f$

    { *source case* }
    (1) $\mathrm{wp}^{src}(\mathcal{E}^{src}.f, \psi, \mathsf{excPost}, \mathtt{m})_{v.f}$
    {*following the definition of the wp function*
    for source expressions in subsection 4.2.2 }

$$\equiv \mathrm{wp}^{src}(\mathcal{E}^{src}, \wedge \begin{array}{l} v \neq \mathbf{null} \Rightarrow \psi \\ \\ v = \mathbf{null} \Rightarrow \mathsf{excPost}(\,\mathtt{NullPntrExc}) \end{array}, \mathsf{excPost}, \mathtt{m})_v$$

    { *bytecode case* }
    (2) $wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{src}.f, e \urcorner, \phi, \ulcorner \mathsf{excPost} \urcorner, \mathtt{m})$
    { *following the definition of the compiler function in subsection 3.3* }

$$\equiv wp^{bc}_{seq}(\begin{array}{l} \ulcorner s, \mathcal{E}^{src}, e - 1 \urcorner \\ e \;\; \mathrm{getfield}\; f \end{array}, \phi, \mathsf{excPost}, \mathtt{m})$$

    {*following the definition of the wp function for bytecode*
    in subsection 5.1 }

$$\equiv wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{src}, e - 1 \urcorner,$$
$$\begin{array}{l} \mathtt{st(cntr\,)}\; \neq \mathbf{null} \Rightarrow \\ \phi[\mathtt{st(\,cntr\,)} \leftarrow \mathtt{st(cntr\,)}.f] \\ \wedge \\ \mathtt{st(cntr\,)}\; = \mathbf{null} \Rightarrow \mathsf{excPost}(\,\mathtt{NullPntrExc}) \end{array},$$
$$\ulcorner \mathsf{excPost} \urcorner, \mathtt{m})$$

    { *From (1) and (2) we apply the induction hypothesis* }
    $\exists Q', R' : \mathcal{F},$

    (3) $\mathrm{wp}^{src}(\mathcal{E}^{src}, \wedge \begin{array}{l} v \neq \mathbf{null} \Rightarrow \psi \\ \\ v = \mathbf{null} \Rightarrow \mathsf{excPost}(\,\mathtt{NullPntrExc}) \end{array}, \mathsf{excPost}, \mathtt{m})_v$

$$\equiv$$

$$Q' \Rightarrow \wedge \begin{array}{l} v \neq \mathbf{null} \Rightarrow \psi \\ \\ v \neq \mathbf{null} \Rightarrow \mathsf{excPost}(\,\mathtt{NullPntrExc}) \end{array}$$
$$\wedge$$
$$R'$$

$$\Rightarrow$$
$$wp^{bc}_{seq}(\ulcorner \mathcal{E}^{src} \urcorner,$$
$$\begin{array}{l} \mathtt{st(cntr\,)}\; \neq \mathbf{null} \Rightarrow \\ \phi[\mathtt{st(\,cntr\,)}\; \leftarrow \mathtt{st(cntr\,)}.f] \\ \wedge \\ \mathtt{st(cntr\,)}\; = \mathbf{null} \Rightarrow \mathsf{excPost}(\,\mathtt{NullPntrExc}) \end{array},$$
$$\ulcorner \mathsf{excPost} \urcorner, \mathtt{m})$$
$$\equiv$$

$$Q' \Rightarrow \begin{array}{l} \texttt{st(cntr)} \neq \mathbf{null} \Rightarrow \phi[\texttt{st(cntr)} \leftarrow \texttt{st(cntr)} .f] \\ \wedge \\ \texttt{st(cntr)} = \mathbf{null} \Rightarrow \mathsf{excPost(\ NullPntrExc)} \end{array} \quad \begin{array}{l} [\texttt{cntr} \leftarrow \texttt{cntr} +1] \\ [\texttt{st(cntr + 1)} \leftarrow v] \end{array}$$

$$\wedge$$
$$R'$$
$$\equiv$$

$\{\ \phi$ *contains only stack expressions* $\texttt{st(cntr - k)}\ , k \geq 0$ *and properties of substitution* $\}$

$$Q' \Rightarrow \begin{array}{l} v \neq \mathbf{null} \Rightarrow \phi \begin{array}{l}[\texttt{cntr} \leftarrow \texttt{cntr} +1] \\ [\texttt{st(cntr + 1)} \leftarrow v.f] \end{array} \\ \wedge \\ v = \mathbf{null} \Rightarrow \mathsf{excPost(\ NullPntrExc)} \end{array}$$

$$\wedge$$
$$R'$$

$\{\ $ *from (3) this case holds* $\}$

Lemma 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* $\phi : \mathcal{F}^{bc}$ *such that* ϕ *may only contain stack expressions of the form* $\texttt{st(cntr - k)}\ , k \geq 0$, *the following holds there exist* $Q, R : \mathcal{F}, v_1 \ldots v_k : \mathcal{E}^{src}$

$$\forall \mathbf{boundVar}_1, \ldots \mathbf{boundVar}_k, Q \Rightarrow$$
$$\psi \begin{array}{l} [v_1 \leftarrow \mathbf{boundVar}_1] \\ \ldots \\ [v_k \leftarrow \mathbf{boundVar}_k] \end{array} \quad \wedge$$
$$R$$
$$\Rightarrow$$

$$wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{src}, e \urcorner, \psi, \mathsf{excPost}, \mathtt{m})\ \equiv$$
$$\forall \mathbf{boundVar}_1, \ldots \mathbf{boundVar}_k, Q \Rightarrow$$
$$\phi \begin{array}{l} [\texttt{cntr} \leftarrow \texttt{cntr} +1] \\ [\texttt{st(cntr +1)} \leftarrow v] \\ [v_1 \leftarrow \mathbf{boundVar}_1] \\ \ldots \\ [v_k \leftarrow \mathbf{boundVar}_k] \end{array}$$

$$\wedge$$
$$R$$

The next lemma establishes a relation between the source and bytecode *wp* w.r.t. to the same weakest precondition.
excPost

Lemma 6.4 (Proof obligation equivalence on statements)

$$\forall \mathcal{STMT}, \psi, \mathsf{excPost}^{src},$$
$$wp^{bc}_{stmt}(\ulcorner s, \mathcal{STMT}, e \urcorner, \psi, \mathsf{excPost}^{src}, \mathtt{m}) =$$
$$wp^{src}(\mathcal{STMT}, \psi, \mathsf{excPost}^{src}, \mathtt{m})$$

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 5.2.4 (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)

The next statement establishes under what conditions the *wp* over source and the *wp*defined in the previous Chapter are equivalent.

Lemma 6.5 (Proof obligation equivalence on statements) *For every* $\mathcal{STMT}$ *compilation* $\ulcorner s, \mathcal{STMT}, e \urcorner$, *formula* ψ *and exceptional postcondition function* $\mathsf{excPost}^{src2bc}$ *such that :*

- $\psi = inter(e, e+1)$

- $\forall \mathtt{Exc}, \mathsf{excPost}^{src2bc}(\mathtt{Exc}) = \mathtt{m}.\mathsf{excPost}(\mathtt{Exc}, e)$

then it holds

$$wp^{src}(\mathcal{STMT}, \psi, \mathsf{excPost}^{src}, \mathtt{m}) = wp(ins_s, \mathtt{m})$$

From the last lemma follows that if the body of the method $\mathtt{m}$is $\mathcal{STMT}$, and its compilation is $\ulcorner s, \mathcal{STMT}, e \urcorner$, then the proof obligations generated over $\mathcal{STMT}$ upon postcondition $\mathtt{m}.\psi^{postN}$ and exceptional postcondition function $\mathtt{m}.exc^{bc}$ and its compilation $\ulcorner s, \mathcal{STMT}, e \urcorner$ are the same

$$\mathrm{wp}^{src}(\mathcal{STMT}, \psi^{postN}, \mathsf{excPost}^{src}, \mathtt{m}) = wp(ins_s, \mathtt{m})$$

References

[1] Lilian Burdy and Mariela Pavlova. From JML to BCSL. Technical report, INRIA, Sophia-Antipolis, 2004. Draft version. Available from `http://www.inria.fr/everest/Mariela.Pavlova`.

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

[3] Mariela Pavlova. Bytecode specification and verification. Technical report, INRIA, Sophia-Antipolis, 2005. Draft version. Available from `http://www.inria.fr/everest/Mariela.Pavlova`.