Compiling Proof Obligations

Mariela Pavlova

July 3, 2006

Contents

1 Introduction

This documents studies the relationship between the verification conditions generated for a Java like source language and the verification conditions generated for the bytecode language defined in [3]. We establish an equivalence which we name $=^{mod\ Names\ and\ bools}$ modulo names and boolean values of the proof obligations on source and bytecode level. This result may have an impact on the application on PCC techniques for complex functional and security properties where full automatisation is not possible.

The traditional PCC architecture comes along with a certifying compiler. The basic idea is that the certifying compiler infers automatically annotations, automatically generates verification conditions, proves them automatically and then sends both the code and the proof certificate to the counterpart that will run the code. The receiver then, generates the verification conditions and type checks the generated formulas against the proof certificate. This architecture works for properties like well typedness and safe memory read/write but it is not applicable for complex policies where the specification and the proof cannot be done automatically.

... v

2 Source

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. 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*

$$\begin{array}{lll}
\mathcal{E}^{src} ::= & \mathbf{constInt} \\
 & \mid \mathbf{true} \\
 & \mid \mathbf{false} \\
 & \mid \mathcal{E}^{src}\ op\ \mathcal{E}^{src} \\
 & \mid \mathcal{E}^{src}.f \\
 & \mid \mathbf{var} \\
 & \mid (Class)\ \mathcal{E}^{src} \\
 & \mid \mathbf{null} \\
 & \mid \mathbf{this} \\
 & \mid \mathcal{E}^{\mathcal{R}} \\
 & \mid \mathcal{E}^{src}.m(\mathcal{E}^{src}) \\
 & \mid \mathbf{new}\ Class(\mathcal{E}^{src})
\end{array}$$

$$\begin{array}{lll}
\mathcal{E}^{\mathcal{R}} ::= & \mathcal{E}^{src}\ \mathcal{R}\ \mathcal{E}^{src} \\
 & \mid \mathcal{E}^{src}\ \mathbf{instanceof}\ Class \\
 & \\
 & \mathcal{R} \in \{\leq, <, \geq, >, =, \neq\}
\end{array}$$

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

- **constInt** is any integer literal
- **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
- $\mathcal{E}^{src}.m(\mathcal{E}^{src})$ stands for a method invokation expression. Note that here we consider only methods with one argument 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

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

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

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

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` $\{\mathcal{STMT}\}$
- `try` $\{\mathcal{STMT}\}$ `catch` (`Exc`) $\{\mathcal{STMT}\}$ `finally` $\{\mathcal{STMT}\}$
- `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 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 function is denoted with $\ulcorner\urcorner$ and its signature is :

$$\ulcorner\urcorner : nat * \mathcal{STMT} * nat \longrightarrow list \;\; \mathrm{I}$$

The compiler function takes three arguments: a natural number s from which the labeling of the compilation of $\mathcal{STMT}$ starts, the compiled statement $\mathcal{STMT}$ and a natural number which is the greatest label in the compilation of $\mathcal{STMT}$ and returns a list of bytecode instructions.

the exception handler function

3.1 Compiling expressions in bytecode instructions

- 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

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

- 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

$$\ulcorner s, \mathbf{var}, s \urcorner = s: \text{ load } \mathtt{reg_i}$$

where $\mathtt{reg_i}$ is the local variable at index i

- arithmetic expressions

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

- cast expression

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

- instanceof expression

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

- null expression

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

- object creation

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

- this instance

$$\ulcorner s, \mathbf{this}, s \urcorner = s: \text{ load } \mathtt{reg_0}$$

3.2 Compiling control statements in bytecode instructions

about the compilation of exception handlers

- compositional statement

$$\begin{array}{l} \ulcorner s, \mathcal{STMT}_1; \mathcal{STMT}_2, e \urcorner = \\ \ulcorner s, \mathcal{STMT}_1, e' \urcorner; \\ \ulcorner e', \mathcal{STMT}_2, e \urcorner \end{array}$$

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

- if statement

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

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

  - field assignement.

$$\begin{array}{l}
\ulcorner s, \mathcal{E}^{src}_1.f = \mathcal{E}^{src}_2, e\urcorner = \\
\ulcorner s, \mathcal{E}^{src}_1, e'\urcorner; \\
\ulcorner e', \mathcal{E}^{src}_2, e-1\urcorner; \\
e: \text{ putfield } f;
\end{array}$$

  - method local variable or parameter update

$$\begin{array}{l}
\ulcorner s, \mathbf{var} = \mathcal{E}^{src}, e\urcorner = \\
\ulcorner s, \mathcal{E}^{src}, e-1\urcorner \\
e: \text{ store } \texttt{reg}_\texttt{i};
\end{array}$$

- try catch statement

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

$$\text{addExcHandler}(s,\; e',\; e'+2,\; Class))$$

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

- try finally statement

$\ulcorner s, \texttt{try}\ \{\mathcal{STMT}_1\}\ \texttt{finally}\ \{\mathcal{STMT}_2\}, e\urcorner =$
$\ulcorner s, \mathcal{STMT}_1, e'\urcorner;$
$e'+1:$ jsr $e'+7$;
$e'+2:$ goto $e+1$;

{ default exception handler}
$e'+3:$ store l;
$e'+4:$ jsr $e'+7$;
$e'+5:$ load l;
$e'+6:$ athrow;

{ compilation of the subroutine}
$e'+7:$ store k;
$\ulcorner e'+8, \mathcal{STMT}_2, e-1\urcorner$
$e:$ ret k

addExcHandler($s,\ e',\ e'+8,\ \mathit{Exception}$))

We keep close to the JVM (short for Java Virtual Machine) specification, which requires that the subroutines must be compiled using jsr and ret instructions. The jsr actually jumps to the first instruction of the compiled subroutine which starts at index s 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 .

Note:

1. we assume that the local variable e and k are not used in the compilation of the statement $\mathcal{STMT}_1$.
2. here we also assume that the statement $\mathcal{STMT}_1$ does not contain a return instruction

The compiler adds a default exception handler whose implementation guarantees that in exceptional termination case, the subroutine is also executed. The exception handler is added in the exception handler table.

- try catch finally statement

$$\ulcorner s, \texttt{try}\ \{\mathcal{STMT}_1\}\ \texttt{catch}\ (\mathit{Class})\ \{\mathcal{STMT}_2\}\ \texttt{finally}\ \{\mathcal{STMT}_3\}, e\urcorner =$$

$$\ulcorner s, \texttt{try}\ \{\texttt{try}\ \{\mathcal{STMT}_1\}\ \texttt{catch}\ (\mathit{Class})\ \{\mathcal{STMT}_2\}\ \}\ \texttt{finally}\ \{\mathcal{STMT}_3\}, e\urcorner$$

- 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

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

- return statement

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

3.3 Properties of the compiler function

In this last subsection, we will look at the properties of the bytecode programs produced by the compiler.

A concept which is useful for the rest of the section is the block of instructions which execute sequentially. The next definition is a formal description.

Definition 3.3.1 (Block of instructions) *If the list of instructions* $l = [i_1 : \texttt{instr} \dots i_k : \texttt{instr}]$ *is such that*

- *none of the instructions in the set is a jump instruction, i.e.* $\forall m, m = 1..k \Rightarrow \neg(i_m : \texttt{instr} \in \{\ \text{goto}\ ,\ \text{if}\})$
- *none of the instructions is a target of an instruction* $i_j : \texttt{instr}$ *which does not belong to* l

We denote such a list of instruction with $i_1 : \texttt{instr}; ...; i_k : \texttt{instr}$

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

Property 3.3.1 (Compilation of expressions) *For any expression* $\mathcal{E}^{src}$*, starting label* s *and end label* e*, the compiler will produce a block of bytecode instruction* $\ulcorner s, \mathcal{E}^{src}, e\urcorner$ *such that* $s : \texttt{instr}; ...; e : \texttt{instr}$

The property states that there are no jump instructions in the list of instructions representing the compilation of an expression. The proof is done by induction over the structure of the expression.

Property 3.3.2 (Compilation of statements) *For any expression* $\mathcal{STMT}$*, start label* s *and end label* e*, the compiler will produce a sequence of bytecode instruction* $\ulcorner s, \mathcal{STMT}, e\urcorner$ *such that:*

$$\begin{array}{l}\forall i, (\mathit{inList}(\ulcorner s, \mathcal{STMT}, e\urcorner, i : \texttt{instr}))\wedge \\ \quad (i : \texttt{instr} \rightarrow k : \texttt{instr})\wedge \\ \quad \neg(\mathit{inList}(\ulcorner s, \mathcal{STMT}, e\urcorner, k : \texttt{instr})) \\ \quad \neg \mathit{isExcHandlerStart}(k : \texttt{instr}) \Rightarrow \\ \qquad k = e+1\end{array}$$

The following property states that if there are instructions inside the compiled statetement $\ulcorner s, \mathcal{STMT}, e \urcorner$ which are in execution relation[1] (but not with the start of an exception handler) with an instruction $k : \texttt{instr}$ outside it then the targetted instruction is $e{+}1 : \texttt{instr}$. The conditions $\neg(\mathit{inList}(\ulcorner s, \mathcal{STMT}, e \urcorner, k : \texttt{instr}))$ and $\neg \mathit{isExcHandlerStart}(k : \texttt{instr})$ eliminate the case when the execution relation is between an instruction inside $\ulcorner s, \mathcal{STMT}, e \urcorner$ which may throw an exception and the start instruction of the proper handler exception handler.

The proof is done by induction on the structure of the compiled statement.

Proof

We scatch the proof for the compilation of the if statement.

{ *by definition of the compiler function for if statements in section 3.2* }
$\ulcorner s, \texttt{if}\ (\mathcal{E}^{\mathcal{R}})\ \texttt{then}\ \{\mathcal{STMT}_1\}\ \texttt{else}\ \{\mathcal{STMT}_2\}, e \urcorner =$
$\ulcorner s, \mathcal{E}^{\mathcal{R}}, e' \urcorner;$
$e' + 1 :\ \text{if } e'' + 2;$
$\ulcorner e' + 2, \mathcal{STMT}_2, e'' \urcorner$
$e'' + 1 :\ \text{goto}\ \ e + 1;$
$\ulcorner e'' + 2, \mathcal{STMT}_1, e \urcorner;$

{ *induction hypothesis for $\mathcal{E}^{\mathcal{R}}$ and $\mathcal{STMT}_1$ and $\mathcal{STMT}_2$* }

(1) $\forall i, s \geq i \leq e',\ i : \texttt{instr} \rightarrow k : \texttt{instr}) \wedge$
$\neg(\mathit{inList}(\ulcorner s, \mathcal{STMT}, e \urcorner, k : \texttt{instr}))$
$\neg \mathit{isExcHandlerStart}(k : \texttt{instr}) \Rightarrow$
$k = e' + 1$

(2) $\forall i, e' + 2 \geq i \leq e'',\ i : \texttt{instr} \rightarrow k : \texttt{instr}) \wedge$
$\neg(\mathit{inList}(\ulcorner s, \mathcal{STMT}, e \urcorner, k : \texttt{instr}))$
$\neg \mathit{isExcHandlerStart}(k : \texttt{instr}) \Rightarrow$
$k = e'' + 1$

(3) $\forall i, e'' + 2 \geq i \leq e,\ i : \texttt{instr} \rightarrow k : \texttt{instr}) \wedge$
$\neg(\mathit{inList}(\ulcorner s, \mathcal{STMT}, e \urcorner, k : \texttt{instr}))$
$\neg \mathit{isExcHandlerStart}(k : \texttt{instr}) \Rightarrow$
$k = e + 1$

(4) { *from (1),(2) and (3) we get that jumps from $\mathcal{E}^{\mathcal{R}}$ and $\mathcal{STMT}_1$ go inside the compilation of* $\texttt{if}\ (\mathcal{E}^{\mathcal{R}})\ \texttt{then}\ \{\mathcal{STMT}_1\}\ \texttt{else}\ \{\mathcal{STMT}_2\}$ *as $e' + 1$ and $e'' + 1$ are labels in the compilation of the statement and and that jumps from $\mathcal{STMT}_2$ go to $e + 1$* }

(5) {*the instruction* $e' + 1 :\ \text{if } e'' + 2;$ *may jump to $e'' + 2$ which is inside the compilation of the if statement* }

(6) { *the instruction* $e'' + 1 :\ \text{goto}\ \ e + 1;$ *may jump to $e + 1$* }

from *(4)*, *(5)* and *(6)* the lemma holds in that case

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

give the definition of the function $\mathit{isExcHandlerStart}(k : \texttt{instr})$

this is not sufficient. There must be established a property that the $\rightarrow^{l}$ is only in the compilation

In the following, we abstract from the labeling done by the compiler and we will denote the compiler function with [ ]. We assume that the compiler function [ ] does a labeling which has the above properties. Also, the notation for labeled instructions used here i : `instr` will be sometimes abbreviated just to the index of the instruction i.

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.

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

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

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

$$\begin{array}{ll}
\mathcal{E}^{spec} ::= & \mathbf{constInt} \\
 & |\ \mathbf{true} \\
 & |\ \mathbf{false} \\
 & |\ \mathbf{ref} \\
 & |\ \mathcal{E}^{spec}\ op\ \mathcal{E}^{spec} \\
 & |\ \mathcal{E}^{spec}.f \\
 & |\ \mathbf{var} \\
 & |\ \mathbf{null} \\
 & |\ \mathbf{this} \\
 & |\ \backslash typeof(\mathcal{E}^{spec}) \\
 & |\quad \backslash \mathrm{result}
\end{array}$$

Note that the expressions in the assertion language are very similar to the expression in the programming language presented in subsection 2.

We define a function which maps expressions from the programming language into the expressions of the assertion language which is denoted and is typed as follows:

$$\ulcorner . \urcorner^{src2spec} : \mathcal{E}^{src} \rightarrow \mathcal{E}^{spec}$$

The function is defined as follows:

$$
\begin{array}{lcl}
\ulcorner \mathbf{constInt} \urcorner^{src2spec} & = & \mathbf{constInt} \\
\ulcorner \mathbf{true} \urcorner^{src2spec} & = & \mathbf{true} \\
\ulcorner \mathbf{false} \urcorner^{src2spec} & = & \mathbf{false} \\
\ulcorner \mathcal{E}^{src}\ op\ \mathcal{E}^{src} \urcorner^{src2spec} & = & \ulcorner \mathcal{E}^{src} \urcorner^{src2spec}\ op\ \ulcorner \mathcal{E}^{src} \urcorner^{src2spec} \\
\ulcorner (Class)\mathcal{E}^{src} \urcorner^{src2spec} & = & \ulcorner \mathcal{E}^{src} \urcorner^{src2spec} \\
\ulcorner \mathcal{E}^{src}.m(\mathcal{E}^{src}) \urcorner^{src2spec} & = & \mathbf{ref} \\
\ulcorner \mathcal{E}^{src}.f \urcorner^{src2spec} & = & \ulcorner \mathcal{E}^{src} \urcorner^{src2spec}.f \\
\ulcorner \mathbf{this} \urcorner^{src2spec} & = & \mathbf{this} \\
\ulcorner \mathbf{new}\ Class(\mathcal{E}^{src}) \urcorner^{src2spec} & = & \mathbf{ref} \\
\ulcorner \mathcal{E}^{src}\ \mathbf{instanceof}\ Class \urcorner^{src2spec} & = & \backslash\mathtt{typeof}(\ulcorner \mathcal{E}^{src} \urcorner^{src2spec}) <: Class \wedge \ulcorner \mathcal{E}^{src} \urcorner^{src2spec} \neq \mathbf{null} \\
\ulcorner \mathcal{E}^{src}\ \mathcal{R}\ \mathcal{E}^{src} \urcorner^{src2spec} & = & \ulcorner \mathcal{E}^{src} \urcorner^{src2spec} \mathcal{R} \ulcorner \mathcal{E}^{src} \urcorner^{src2spec}
\end{array}
$$

4.2 Weakest Predicate Transformer for the Source Language

The weakest precondition calculates for every statement $\mathcal{STMT}$ from our source language, for any normal postcondition $Post$ and exceptional postcondition function ePost^{src} ($\mathtt{Exc} \rightarrow \mathcal{STMT} \rightarrow \mathcal{F}^{src}$), 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 $\mathsf{ePost}^{src}(Exc, \mathcal{STMT})$ holds. The weakest precondition function has the following signature:

$$\mathrm{wp}^{src} : \mathcal{STMT} \rightarrow \mathcal{F}^{src} \rightarrow (\ \mathtt{Exc} \rightarrow \mathcal{F}^{src}) \rightarrow \mathcal{F}^{src}$$

Before looking at the definition of the weakest predicate transformer we define the exceptional postcondition function ePost^{src}.

4.2.1 Exceptional Postcondition Function

We now look at how the exceptional postconditions for expressions(statements) are managed. As we said the weakest predicate transformer takes into account 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 ePost^{src} which maps exception types to predicates

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

The function ePost^{src} returns the predicate $\mathsf{ePost}^{src}(\mathtt{Exc})$ 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 ePost^{src} which are defined in the usual way

$$\mathsf{ePost}^{src}[\oplus\mathtt{Exc'} \rightarrow P](\mathtt{Exc}, exp) = \begin{cases} P & if \mathtt{Exc} <: \mathtt{Exc'} \\ \mathsf{ePost}^{src}(\mathtt{Exc}, exp) & else \end{cases}$$

4.2.2 Expressions

We define the weakest precondition predicate transformer function over expressions. As we will see in the definition below this definition allows us to get the side effect conditions of the expression evaluationm, namely the conditions for normal and exceptional termination.

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

$$\mathrm{wp}^{src}(\ const\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src}) = \mathsf{nPost}^{src}$$

- field access expression

$$\begin{array}{l}
\mathrm{wp}^{src}(\ \mathcal{E}_1^{src}.f\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src}) = \\
\mathrm{wp}^{src}(\ \mathcal{E}_1^{src}\ , \\
\qquad \left.\begin{array}{l}
\ulcorner\mathcal{E}_1^{src}\urcorner^{src2spec} \neq \textbf{null} \Rightarrow \mathsf{nPost}^{src} \\
\wedge \\
\ulcorner\mathcal{E}_1^{src}\urcorner^{src2spec} = \textbf{null} \Rightarrow \mathsf{ePost}^{src}(\ \mathtt{NullPointerExc}, \mathcal{E}_1^{src})
\end{array}\right. , \\
\qquad \mathsf{ePost}^{src})
\end{array}$$

- arithmetic expressions

$$\begin{array}{l}
\mathrm{wp}^{src}(\ \mathcal{E}_1^{src}\ op\ \mathcal{E}_2^{src}\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src}) = \\
\mathrm{wp}^{src}(\ \mathcal{E}_1^{src}\ , \mathrm{wp}^{src}(\ \mathcal{E}_2^{src}\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src}), \mathsf{ePost}^{src})
\end{array}$$

- method invocation

$$\begin{array}{l}
\mathrm{wp}^{src}(\ \mathcal{E}_1^{src}.m(\mathcal{E}_2^{src})\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src}) = \\
\mathrm{wp}^{src}(\ \mathcal{E}_1^{src}\ , \mathrm{wp}^{src}(\ \mathcal{E}_2^{src}\ , \\
\qquad \left\{\begin{array}{l}
\mathcal{E}_1^{src} \neq\ \textbf{null} \Rightarrow \\
\qquad m.\mathsf{Pre}^{src}\ \begin{array}{l}[\textbf{this} \leftarrow \mathcal{E}_1^{src}] \\ [\mathrm{arg} \leftarrow \mathcal{E}_2^{src}]\end{array} \\
\qquad \wedge \\
\qquad \forall \textbf{ref},\ \forall\ m \in m.\mathsf{modif}^{src} \\
\qquad \left\{\begin{array}{l}
\mathtt{\backslash typeof(}\textbf{ref}\mathtt{)} <: m.\mathsf{retType} \wedge \\
m.\mathsf{nPost}^{src}\ \begin{array}{l}[\ \ \backslash\mathrm{result}\ \ \leftarrow \textbf{ref}] \\ [\textbf{this} \leftarrow \mathcal{E}_1^{src}] \\ [\mathrm{arg} \leftarrow \mathcal{E}_2^{src}]\end{array} \\
\qquad \Rightarrow \mathsf{nPost}^{src}[\ulcorner\mathcal{E}_1^{src}.m(\mathcal{E}_2^{src})\urcorner^{src2spec} \leftarrow \textbf{ref}]
\end{array}\right. \\
\qquad \wedge \\
\qquad \forall \mathtt{E} \in m.\mathsf{exceptions}^{src}, \\
\qquad \forall\ m \in m.\mathsf{modif}^{src} \\
\qquad\ m.\mathsf{exc}^{src}(\mathtt{E}) \Rightarrow \mathsf{ePost}^{src}(\mathtt{E}) \\
\mathcal{E}_1^{src} =\ \textbf{null} \Rightarrow \mathsf{ePost}^{src}(\ \mathtt{NullPntrExc})
\end{array}\right. , \\
\qquad \mathsf{ePost}^{src}), \\
\mathsf{ePost}^{src})
\end{array}$$

$$where\ \ulcorner\mathcal{E}_1^{src}.m(\mathcal{E}_2^{src})\urcorner^{src2spec} = \textbf{ref}$$

- Cast expression

may be give an example

$$\begin{array}{l}
\mathrm{wp}^{src}(\ (\ \texttt{Class}\ )\ \mathcal{E}^{src}\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src}) = \\
\mathrm{wp}^{src}(\ \mathcal{E}^{src}\ , \\
\quad \texttt{\textbackslash typeof}(\ulcorner \mathcal{E}^{src} \urcorner^{src2spec}) <:\ \texttt{Class}\ \Rightarrow \\
\qquad\qquad \mathrm{wp}^{src}(\ \mathcal{E}^{src}\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src}) \\
\quad \wedge \qquad\qquad\qquad\qquad\qquad\qquad\qquad , \\
\quad \neg\ \texttt{\textbackslash typeof}(\ulcorner \mathcal{E}^{src} \urcorner^{src2spec}) <:\ \texttt{Class}\ \Rightarrow \\
\qquad\qquad \mathsf{ePost}^{src}(\ \texttt{CastExc}, \mathcal{E}^{src}) \\
\quad \mathsf{ePost}^{src})
\end{array}$$

- Null expression

$$\mathrm{wp}^{src}(\ \mathbf{null}\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src}) = \mathsf{nPost}^{src}$$

- this

$$\mathrm{wp}^{src}(\ \mathbf{this}\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src}) = \mathsf{nPost}^{src}$$

- instance creation

$$\begin{array}{l}
\mathrm{wp}^{src}(\ \mathbf{new}\ Class(\mathcal{E}^{src})\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src}) = \\
\mathrm{wp}^{src}(\ \mathcal{E}^{src}\ , \\
\qquad \left\{ \begin{array}{l}
\mathsf{constr}(Class).\mathsf{Pre}^{src}\ \ [arg \leftarrow \ulcorner \mathcal{E}^{src} \urcorner^{src2spec}] \\
\wedge \\
\forall \mathbf{ref}, \\
\quad \wedge \\
\quad not\ \mathbf{instances}(\mathbf{ref}) \wedge \\
\quad \mathbf{instances}(\mathbf{ref}) \neq \mathbf{null} \wedge \\
\quad \forall\ m \in \mathsf{constr}(Class).\mathsf{modif}^{src}, \\
\qquad \mathsf{constr}(Class).\mathsf{nPost}^{src} \Rightarrow \mathsf{nPost}^{src}\ \begin{array}{l} [\mathbf{this} \leftarrow \mathbf{ref}] \\ [arg \leftarrow \ulcorner \mathcal{E}^{src} \urcorner^{src2spec}] \\ [\texttt{\textbackslash typeof}(\mathbf{ref}) \leftarrow Class] \end{array} \\
\wedge \\
\forall \mathsf{Exc} \in \mathsf{constr}(Class).\mathsf{exceptions}^{src}, \\
\forall\ m \in \mathsf{constr}(Class).\mathsf{modif}^{src}, \\
\mathsf{constr}(Class).\mathsf{exc}^{src}(\texttt{Exc}) \Rightarrow \mathsf{ePost}^{src}(\texttt{Exc})
\end{array} \right. , \\
\quad \mathsf{ePost}^{src})
\end{array}$$

$$where\ \ulcorner \mathbf{new}\ Class(\mathcal{E}^{src}) \urcorner^{src2spec} = \mathbf{ref}$$

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

- Instanceof expression

$$\begin{array}{l}
\mathrm{wp}^{src}(\ \mathcal{E}^{src}\ instanceof\ Class\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src}) = \\
\mathrm{wp}^{src}(\ \mathcal{E}^{src}\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src})
\end{array}$$

- Binary relation over expressions

$$\begin{array}{l}\mathrm{wp}^{src}(\ \mathcal{E}_1^{src}\ \mathcal{R}\ \mathcal{E}_2^{src}\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src}) = \\ \mathrm{wp}^{src}(\ \mathcal{E}_1^{src}\ , \mathrm{wp}^{src}(\ \mathcal{E}_2^{src}\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src}), \mathsf{ePost}^{src})\end{array}$$

4.2.3 Statements

- integer and boolean constant access

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

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

- assignment
  - local variable assignemnt

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

$$\begin{array}{l}\mathrm{wp}^{src}(\ \mathcal{E}_2^{src}\ , \\ \qquad \mathrm{wp}^{src}(\ \mathcal{E}_1^{src}\ , \mathsf{nPost}^{src}[\ulcorner \mathcal{E}_1^{src} \urcorner^{src2spec} \leftarrow \ulcorner \mathcal{E}_2^{src} \urcorner^{src2spec}], \mathsf{ePost}^{src}), \\ \qquad \mathsf{ePost}^{src})\end{array}$$

  - instance field assignemnt

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

$$\begin{array}{l}\mathrm{wp}^{src}(\ \mathcal{E}_1^{src}\ , \\ \qquad \mathrm{wp}^{src}(\ \mathcal{E}_2^{src}\ ,\ \begin{array}{l} \mathbf{null} \neq \ulcorner \mathcal{E}_1^{src} \urcorner^{src2spec} \Rightarrow \\ \qquad \mathsf{nPost}^{src}[f \leftarrow f \oplus [\ulcorner \mathcal{E}_1^{src} \urcorner^{src2spec} \rightarrow \ulcorner \mathcal{E}_2^{src} \urcorner^{src2spec}]] \\ \wedge \\ \mathbf{null} = \ulcorner \mathcal{E}_1^{src} \urcorner^{src2spec} \Rightarrow \\ \qquad \mathsf{ePost}^{src}(\texttt{NullPointerExc}) \end{array} , \\ \qquad \mathsf{ePost}^{src}), \\ \qquad \mathsf{ePost}^{src})\end{array}$$

- if statement

$$\mathrm{wp}^{src}(\ \begin{array}{l}\texttt{if } (\mathcal{E}^{src}) \\ \texttt{then}\{\mathcal{STMT}_1\} \\ \texttt{else } \{\mathcal{STMT}_2\}\end{array}\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src}) =$$

$$\begin{array}{l}\mathrm{wp}^{src}(\ \mathcal{E}^{\mathcal{R}}\ , \\ \qquad \begin{array}{l} \ulcorner \mathcal{E}^{\mathcal{R}} \urcorner^{src2spec} \Rightarrow \mathrm{wp}^{src}(\ \mathcal{STMT}_1\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src}) \\ \wedge \\ \neg \ulcorner \mathcal{E}^{\mathcal{R}} \urcorner^{src2spec} \Rightarrow \mathrm{wp}^{src}(\ \mathcal{STMT}_2\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src}) \end{array} , \\ \qquad \mathsf{ePost}^{src})\end{array}$$

- throw exceptions

$$\mathrm{wp}^{src}(\ \texttt{throw}\ \mathcal{E}^{src}\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src}) =$$

$$\begin{array}{l}\mathrm{wp}^{src}(\ \mathcal{E}^{src}\ , \\ \qquad \begin{array}{l} \ulcorner \mathcal{E}^{src} \urcorner^{src2spec} \neq \mathbf{null} \Rightarrow \mathsf{ePost}^{src}(\texttt{\textbackslash typeof}(\mathcal{E}^{src})) \\ \ulcorner \mathcal{E}^{src} \urcorner^{src2spec} = \mathbf{null} \Rightarrow \mathsf{ePost}^{src}(\texttt{NullPointerExc}) \end{array} , \\ \qquad \mathsf{ePost}^{src})\end{array}$$

- try catch statement

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

$$\begin{array}{l} \mathrm{wp}^{src}(\ \mathcal{STMT}_1\ , \\ \quad \mathsf{nPost}^{src}, \\ \quad \mathsf{ePost}^{src} \oplus [\texttt{Exc} \longrightarrow \mathrm{wp}^{src}(\ \mathcal{STMT}_2\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src})]) \end{array}$$

- try finally

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

$$\begin{array}{l} \mathrm{wp}^{src}(\ \mathcal{STMT}_1\ , \\ \quad \mathrm{wp}^{src}(\ \mathcal{STMT}_2\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src}), \\ \quad \mathsf{ePost}^{src} \oplus [\texttt{Exception} \longrightarrow \mathrm{wp}^{src}(\ \mathcal{STMT}_2\ , \mathsf{ePost}^{src}(\texttt{Exception}), \mathsf{ePost}^{src})]) \end{array}$$

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

- try catch finally

$$\mathrm{wp}^{src}(\begin{array}{l} \texttt{try}\ \{\mathcal{STMT}_1\} \\ \texttt{catch}(Class\ c)\ \{\mathcal{STMT}_2\} \\ \texttt{finally}\ \{\mathcal{STMT}_3\} \end{array}\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src})$$

$$=$$

$$\mathrm{wp}^{src}(\begin{array}{l} \texttt{try}\ \{\texttt{try}\ \{\mathcal{STMT}_1\}\texttt{catch}(Class\ c)\ \{\mathcal{STMT}_2\}\} \\ \texttt{finally}\ \{\mathcal{STMT}_3\} \end{array}\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src})$$

- loop statement

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

$$\begin{array}{l} \texttt{INV} \wedge \\ \forall\ m, m \in \texttt{modif}, \\ \quad \texttt{INV} \Rightarrow \\ \quad\quad \mathrm{wp}^{src}(\ \mathcal{E}^{src}\ , \\ \quad\quad\quad \begin{array}{l} \ulcorner \mathcal{E}^{src} \urcorner^{src2spec} = \textbf{true} \Rightarrow\ \mathrm{wp}^{src}(\ \mathcal{STMT}\ , \texttt{INV}, \mathsf{ePost}^{src}) \\ \ulcorner \mathcal{E}^{src} \urcorner^{src2spec} = \textbf{false} \Rightarrow \mathsf{nPost}^{src} \end{array}\ , \\ \quad\quad \mathsf{ePost}^{src}) \end{array}$$

- return statement

$$\begin{array}{l} \mathrm{wp}^{src}(\ \texttt{return}\ \mathcal{E}^{src}\ , \mathsf{nPost}^{src}, \mathsf{ePost}^{src}) = \\ \mathrm{wp}^{src}(\ \mathcal{E}^{src}\ , \mathsf{nPost}^{src}[\ \ \backslash\mathrm{result}\ \leftarrow \ulcorner \mathcal{E}^{src} \urcorner^{src2spec}], \mathsf{ePost}^{src}) \end{array}$$

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 predicate transformer for Bytecode language

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 instruction which takes explicitely the postcondition and the exceptional postcondition function upon which the precondition will be calculated. Its signature is the following:

$$\mathrm{wp}^{bc} : \ \mathrm{I} \rightarrow \mathcal{F}^{bc} \rightarrow (\ \mathtt{Exc} \rightarrow \mathcal{F}^{bc}) \rightarrow \mathcal{F}^{bc}$$

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

$$\begin{array}{l} wp^{bc}(\text{ getfield } f, \psi, \psi^{bc}_{exc}, \mathtt{m}) = \\ \mathtt{st(cntr\ -1\ )} \neq \mathbf{null} \Rightarrow \psi[\mathtt{st(cntr\ )} \leftarrow f(\mathtt{st(cntr\ )})] \wedge \\ \mathtt{st(cntr\ -1\ )} = \mathbf{null} \Rightarrow \psi^{bc}_{exc}(\ \mathtt{NullPntrExc}) \end{array}$$

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 sequence of instruction that always execute sequentially as follows:

Definition 5.1 (*wp* for a block of instructions)

$$\begin{array}{l} wp^{bc}_{seq}(1 : \mathtt{instr}; ...; k : \mathtt{instr}, \psi, \psi^{bc}_{exc}, \mathtt{m}) = \\ wp^{bc}_{seq}(1 : \mathtt{instr}; ...; k-1 : \mathtt{instr}, wp^{bc}(k : \mathtt{instr}, \psi, \psi^{bc}_{exc}, \mathtt{m}), \psi^{bc}_{exc}, \mathtt{m}) \end{array}$$

We turn now to the rules for compiled expressions. Note that from Property 3.3.2 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 5.2 (*wp* for compiled expressions) *For any expression $\mathcal{E}^{src}$, postcondition ψ and exceptional postcondition function ψ^{bc}_{exc} the wp function for the compilation $\ulcorner \mathcal{E} \urcorner$ is $wp^{bc}_{seq}(\ulcorner \mathcal{E} \urcorner, \psi, \psi^{bc}_{exc}, \mathtt{m})$*

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

$$\begin{array}{l} \mathcal{E}^{src}_1; \\ \text{getfield } f \end{array}$$

produce the following formula

$$wp^{bc}_{seq}(\begin{array}{l} \ulcorner \mathcal{E}^{src} \urcorner; \\ \text{getfield } f \end{array} , \psi, \psi^{bc}_{exc}, \mathtt{m})$$

This is equivalent to :

$$wp^{bc}_{seq}(\ulcorner \mathcal{E}^{src} \urcorner, wp^{bc}(\text{ getfield } f, \psi, \psi^{bc}_{exc}, \mathtt{m}), \psi^{bc}_{exc}, \mathtt{m})$$

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

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

The definition of wp^{bc}_{stmt} uses the compiler function defined in Section 3.2

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

$$\begin{array}{l} wp^{bc}_{stmt}(\ulcorner \mathcal{STMT}_1; \mathcal{STMT}_2 \urcorner, \psi, \psi^{bc}_{exc}, \mathtt{m}) =^{def} \\ wp^{bc}_{stmt}(\ulcorner \mathcal{STMT}_1 \urcorner; , \\ \qquad wp^{bc}_{stmt}(\ulcorner \mathcal{STMT}_2 \urcorner, \psi, \psi^{bc}_{exc}, \mathtt{m}), \\ \qquad \psi^{bc}_{exc}, \mathtt{m}) \end{array}$$

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

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

$$\begin{array}{l} wp^{bc}_{seq}(\ulcorner \mathcal{E}^{\mathcal{R}} \urcorner; \\ , \\ \qquad \begin{array}{l} \mathcal{R}(\texttt{st(cntr) ,st(cntr - 1))} \Rightarrow \\ \qquad wp^{bc}_{stmt}(\ulcorner \mathcal{STMT}_1 \urcorner, \psi, \psi^{bc}_{exc}, \mathtt{m})[t \leftarrow t-2] \\ \wedge \\ \neg\mathcal{R}(\texttt{st(cntr) ,st(cntr - 1))} \Rightarrow \\ \qquad wp^{bc}_{stmt}(\ulcorner \mathcal{STMT}_2 \urcorner, \psi, \psi^{bc}_{exc}, \mathtt{m})[t \leftarrow t-2] \end{array} , \\ \qquad \psi^{bc}_{exc}, \mathtt{m}) \end{array}$$

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

$$\begin{array}{l} wp^{bc}_{stmt}(\ulcorner \mathcal{E}^{src}_1 .f = \mathcal{E}^{src}_2 \urcorner, \psi, \psi^{bc}_{exc}, \mathtt{m}) =^{def} \\ wp^{bc}_{seq}(\ulcorner \mathcal{E}^{src}_1 \urcorner, \\ \qquad wp^{bc}_{seq}(\begin{array}{l} \ulcorner \mathcal{E}^{src}_2 \urcorner; \\ \text{putfield } f \end{array} , \psi, \psi^{bc}_{exc}, \mathtt{m}), \\ \qquad \psi^{bc}_{exc}, \mathtt{m}) \end{array}$$

- try catch statement compilation

$$\begin{array}{l} wp^{bc}_{stmt}(\ulcorner \begin{array}{l} \texttt{try } \{\mathcal{STMT}_1\} \\ \texttt{catch } (\text{ExcClass } \mathbf{var})\{\mathcal{STMT}_2\} \end{array} \urcorner, \psi, \psi^{bc}_{exc}, \mathtt{m}) =^{def} \\ wp^{bc}_{stmt}(\ulcorner \mathcal{STMT}_1 \urcorner, \\ \qquad \psi, \\ \qquad \psi^{bc}_{exc}[\oplus \texttt{ExcClass} \rightarrow wp^{bc}_{stmt}(\ulcorner \mathcal{STMT}_2 \urcorner, \psi, \psi^{bc}_{exc}, \mathtt{m})], \mathtt{m}) \end{array}$$

- try finally statement compilation $\ulcorner$`try` $\{\mathcal{STMT}_1\}$ `finally` $\{\mathcal{STMT}_2\}\urcorner$

$$\begin{array}{l} wp^{bc}_{stmt}(\ulcorner \begin{array}{l}\texttt{try } \{\mathcal{STMT}_1\} \\ \texttt{finally } \{\mathcal{STMT}_2\}\end{array} \urcorner, \psi, \psi^{bc}_{exc}, \texttt{m}) =^{def} \\ wp^{bc}_{stmt}(\ulcorner \mathcal{STMT}_1 \urcorner, \\ \quad wp^{bc}_{stmt}(\ulcorner \mathcal{STMT}_2 \urcorner, \psi, \psi^{bc}_{exc}, \texttt{m}), \\ \quad \psi^{bc}_{exc}[\oplus \texttt{ExcClass} \rightarrow wp^{bc}_{stmt}(\ulcorner \mathcal{STMT}_2 \urcorner, \psi, \psi^{bc}_{exc}, \texttt{m})], \texttt{m}) \end{array}$$

- throw exception compilation $\ulcorner$`throw` $\mathcal{E}^{src}\urcorner$

$$\begin{array}{l} wp^{bc}_{stmt}(\ulcorner \texttt{throw } \mathcal{E}^{src} \urcorner, \psi, \psi^{bc}_{exc}, \texttt{m}) =^{def} \\ wp^{bc}_{seq}(\mathcal{E}^{src}, wp^{bc}(\text{ athrow}, \psi, \psi^{bc}_{exc}, \texttt{m}), \psi^{bc}_{exc}, \texttt{m}) \end{array}$$

- loop statement

5.1 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 two statements give the formalization of the relation between the two calculus.

Lemma 5.1.1 (*wp* for a block of instructions) *For every block of instructions $i_1; \ldots ; i_k$ in method* m *if $\psi = wp(next(i_k)\ , \texttt{m})$ then the following holds*

$$wp^{bc}_{seq}(i_1; \ldots ; i_k, \psi, \psi^{bc}_{exc}, \texttt{m}) = wp(i_1\ , \texttt{m})$$

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

Lemma 5.1.2 (*wp* for compiled expressions) *For every compiled expression $\ulcorner s, \mathcal{E}^{src}, e \urcorner$ in method* m *if $\psi = wp(next(e : \texttt{instr})\ , \texttt{m})$ then the following holds*

$$wp^{bc}_{seq}(\ulcorner s, \mathcal{E}^{src}, e \urcorner, \psi, \psi^{bc}_{exc}, \texttt{m}) = wp(s : \texttt{instr}\ , \texttt{m})$$

Proof: From Property 3.3.1 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 5.1.1 and we get the result.

6 Auxiliary Properties

Before stating the main theorem we need some auxiliary properties. First, we establish that adding a goto instruction to a sequence of instructions does not change the weakest predicate of the augmented bytecode sequence.

Lemma 1 *Let's have the sequence of bytecode instructions $i_1; ...; i_k$ where $next(i_k) = i_l$*

$$wp^{bc}_{seq}(i_1; ...; i_k, \psi, \psi^{bc}_{exc}, \texttt{m}) = wp^{bc}_{seq}(i_1; ...; i_k;\ \text{goto} l, \psi, \psi^{bc}_{exc}, \texttt{m})$$

The proof is based on the fact that the instruction goto *does not have side effects and thus, the following holds:* $wp^{bc}_{seq}(\text{ goto l}, \psi, \psi^{bc}_{exc}, \texttt{m}) = \psi$

We now turn to see how the execution of the compilation $\ulcorner\mathcal{E}^{src}\urcorner$ of an expression $\mathcal{E}^{src}$ affects the operand stack. In particular, we claim that if the execution of the compiled expression $\ulcorner\mathcal{E}^{src}\urcorner$ terminates normally then the stack top contains the value of the expression $\ulcorner\mathcal{E}^{src}\urcorner^{spec}$. This actually reflects how we expect that the virtual machine execute bytecode programs.

This fact in terms of weakest preconditons can be expressed as follows:

Lemma 2 (Wp of a compiled expression) *For any expression $\mathcal{E}^{src}$ from our source language, for any formula $\psi : \mathcal{F}^{src}$ 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)` *, $k \geq 0$, there exist $Q, R : \mathcal{F}^{src}$ such that the following holds*

- $$\begin{array}{l} wp^{src}(\ \mathcal{E}^{src}\ , \psi, \psi^{bc}_{exc}) \ \equiv \\ Q \Rightarrow \psi \\ \wedge \\ R \end{array}$$

- $$\begin{array}{l} wp^{bc}_{seq}(\ulcorner\mathcal{E}^{src}\urcorner, \psi, \ulcorner\psi^{bc}_{exc}\urcorner, \mathtt{m}) \ \equiv \\ \ulcorner Q\urcorner^{spec} \Rightarrow \phi \begin{array}{l}[\mathtt{cntr} \leftarrow \mathtt{cntr} + 1] \\ [\mathtt{st(cntr+1)} \leftarrow \ulcorner\mathcal{E}^{src}\urcorner^{spec}]\end{array} \\ \\ \wedge \\ \ulcorner R\urcorner^{spec} \end{array}$$

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)$\mathrm{wp}^{src}(\ const\ , \psi, \psi^{bc}_{exc})$
$\{$ *following the definition of the wp function for source expressions in subsection 4.2* $\}$
$\equiv \psi$

$\{$ *bytecode case* $\}$
(2)$wp^{bc}_{seq}(\ulcorner const\urcorner, \phi, \ulcorner\psi^{bc}_{exc}\urcorner, \mathtt{m})$
$\{$ *following the definition of the compiler function in subsection 3.1* $\}$
$\equiv wp^{bc}_{seq}(\ \mathrm{push}\ulcorner const\urcorner^{spec}, \phi, \ulcorner\psi^{bc}_{exc}\urcorner, \mathtt{m})$
$\{$ *following the definition of the wp function for bytecode in subsection 5* $\}$
$$\equiv \phi \begin{array}{l}[\mathtt{cntr} \leftarrow \mathtt{cntr} + 1] \\ [\mathtt{st(\ cntr+1)} \leftarrow \ulcorner const\urcorner^{spec}]\end{array}$$

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

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

$\{$ *source case* $\}$

(1) $\mathrm{wp}^{src}(\ \mathcal{E}^{src}.f\ , \psi, \psi^{bc}_{exc})$

$\{$*following the definition of the wp function for source expressions in subsection 4.2* $\}$

$$\equiv \mathrm{wp}^{src}(\ \mathcal{E}^{src}\ ,\ \begin{array}{l} \ulcorner\mathcal{E}^{src}\urcorner^{src2spec} \neq \mathbf{null} \Rightarrow \psi \\ \wedge \\ \ulcorner\mathcal{E}^{src}\urcorner^{src2spec} \neq \mathbf{null} \Rightarrow \psi^{bc}_{exc}(\ \mathtt{NullPntrExc}) \end{array} , \psi^{bc}_{exc})$$

$\{$ *bytecode case* $\}$

(2) $wp^{bc}_{seq}(\ulcorner\mathcal{E}^{src}.f\urcorner, \phi, \ulcorner\psi^{bc}_{exc}\urcorner, \mathtt{m})$

$\{$ *following the definition of the compiler function in subsection 3.1* $\}$

$$\equiv wp^{bc}_{seq}(\ \begin{array}{l}\ulcorner\mathcal{E}^{src}\urcorner; \\ \text{getfield } f\end{array}\ , \phi, \psi^{bc}_{exc}, \mathtt{m})$$

$\{$*following the definition of the wp function for bytecode in subsection 5* $\}$

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

$\{$ *from (1) and (2) we apply the induction hypothesis* $\}$

$\exists Q', R' : \mathcal{F}^{src},$

$$\textit{(3)}\ \mathrm{wp}^{src}(\ \mathcal{E}^{src}\ ,\ \begin{array}{l} \ulcorner\mathcal{E}^{src}\urcorner^{src2spec} \neq \mathbf{null} \Rightarrow \psi \\ \wedge \\ \ulcorner\mathcal{E}^{src}\urcorner^{src2spec} \neq \mathbf{null} \Rightarrow \psi^{bc}_{exc}(\ \mathtt{NullPntrExc}) \end{array} , \psi^{bc}_{exc})$$

$\equiv$

$$Q' \Rightarrow \begin{array}{l} \ulcorner\mathcal{E}^{src}\urcorner^{src2spec} \neq \mathbf{null} \Rightarrow \psi \\ \wedge \\ \ulcorner\mathcal{E}^{src}\urcorner^{src2spec} \neq \mathbf{null} \Rightarrow \psi^{bc}_{exc}(\ \mathtt{NullPntrExc}) \end{array}$$

$\wedge$

R'

$$\textit{(4)}\ wp^{bc}_{seq}(\ulcorner\mathcal{E}^{src}\urcorner, \begin{array}{l} \mathtt{st(cntr\ )} \neq \mathbf{null} \Rightarrow \\ \phi[\mathtt{st(\ cntr\ )} \leftarrow f(\mathtt{st(cntr\ )})] \\ \wedge \\ \mathtt{st(cntr\ )} = \mathbf{null} \Rightarrow \ulcorner\psi^{bc}_{exc}\urcorner(\ \mathtt{NullPntrExc}) \end{array} , \ulcorner\psi^{bc}_{exc}\urcorner, \mathtt{m})$$

$\equiv$

$$\ulcorner Q'\urcorner \Rightarrow \begin{array}{l} \mathtt{st(cntr\ )} \neq \mathbf{null} \Rightarrow \phi[\mathtt{st(\ cntr\ )} \leftarrow f(\mathtt{st(cntr\ )})] \\ \wedge \\ \mathtt{st(cntr\ )} \neq \mathbf{null} \Rightarrow \ulcorner\psi^{bc}_{exc}\urcorner(\ \mathtt{NullPntrExc}) \end{array} \quad \begin{array}{l} [\mathtt{cntr} \leftarrow \mathtt{cntr} + 1] \\ [\mathtt{st(cntr + 1)} \leftarrow \ulcorner\mathcal{E}^{src}\urcorner^{spec}] \end{array}$$

$\wedge$

$\ulcorner R'\urcorner$

$\equiv$

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

$$\ulcorner Q' \urcorner \Rightarrow \begin{array}{l} \ulcorner \mathcal{E}^{src} \urcorner^{spec} \neq \textbf{null} \Rightarrow \phi \begin{array}{l}[\texttt{cntr} \leftarrow \texttt{cntr} + 1] \\ [\texttt{st(cntr + 1)} \leftarrow \ulcorner f(\mathcal{E}^{src}) \urcorner^{spec}]\end{array} \\ \wedge \\ \ulcorner \mathcal{E}^{src} \urcorner^{spec} \neq \textbf{null} \Rightarrow \ulcorner \psi^{bc}_{exc} \urcorner(\texttt{NullPntrExc}) \end{array}$$
$$\wedge$$
$$\ulcorner R' \urcorner$$

$\{$ *from (3) and (4) this case holds* $\}$

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`.