Compiling Proof Obligations

Mariela Pavlova

July 24, 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 **??** presents the weakest precondition predicate transformer which we define over the source language.

Section **??** 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 **??**, 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 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*

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

$$
\begin{aligned}
\mathcal{E}^{\mathcal{R}} ::= \quad & \mathcal{E}^{src}\ \mathcal{R}\ \mathcal{E}^{src} \\
& \mid \mathcal{E}^{src}\ \textbf{instanceof}\ \mathit{Class} \\
& \\
& \mathcal{R} \in \{\leq, <, \geq, >, =, \neq\}
\end{aligned}
$$

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
- $\mathcal{E}^{src}.m()$ 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

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

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

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 :*

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

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` $(\mathcal{E}^{\mathcal{R}})$[`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 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 exception 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:

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

The definition of the procedure is the following:

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

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

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

3.2 Compiling loop invariants

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

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

$$\mathsf{addLoopSpec} : nat \rightarrow P_{bml} \rightarrow list\ E_{bml} \rightarrow \mathbf{Method}$$

The definition of the procedure is hereafter:

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

3.3 Compiling expressions in bytecode instructions

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

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

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

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

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

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

$$\ulcorner s, \mathbf{var}, s \urcorner = s: \text{ load } \mathbf{reg}(i)$$

where $\mathbf{reg}(i)$ is the local variable at index i

- arithmetic expressions

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

- 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

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

- 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

$$\ulcorner s, \mathbf{this}, s \urcorner = s: \text{ load } \mathbf{reg}(0)$$

3.4 Compiling control statements in bytecode instructions

- compositional statement

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

about the compilation of exception handlers

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

- if statement

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

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

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

  - method local variable or parameter update

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

- try catch statement

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

$$\mathsf{addExcHandler}(s,\ e',\ e'+2,\ \mathit{ExcType}, \texttt{m}))$$

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

- try finally statement

$$\begin{array}{l}
\ulcorner s, \texttt{try}\ \{\mathcal{STMT}_1\}\ \texttt{finally}\ \{\mathcal{STMT}_2\}, e\urcorner = \\
\ulcorner s, \mathcal{STMT}_1, e'\urcorner; \\
\ulcorner e'+1, \mathcal{STMT}_2, e''\urcorner; \\
e''+1:\ \text{goto}\ \ e+1; \\
\\
\{\ \text{default exception handler}\} \\
e''+2:\ \text{store}\ l; \\
\ulcorner e''+3, \mathcal{STMT}_2, e-2\urcorner; \\
e-1:\ \text{load}\ l; \\
e:\ \text{athrow};
\end{array}$$

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

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

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

- return statement

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

3.5 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 $e : \texttt{instr}$ in the compilation $\ulcorner s, \mathcal{STMT}, e\urcorner$ of a statement $\mathcal{STMT}$ is always in execution relation with the instruction $e+1 : \texttt{instr}$

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*

$$e : \texttt{instr} \rightarrow e+1 : \texttt{instr}$$

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 $k : \texttt{instr}$ 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 $\neg(\mathit{inList}(\ulcorner s, \mathcal{S}, e\urcorner, k : \texttt{instr})\ )$ and
$\neg \mathit{isExcHandlerStart}(k : \texttt{instr})$ eliminate the case when the execution relation is between an instruction inside $\ulcorner s, \mathcal{S}, e\urcorner$ 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:*

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

[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* $\exists, s \leq i \leq e, \exists k, k \notin [s \ldots e+1] \wedge i : \texttt{instr} \rightarrow k : \texttt{instr}$ }

{ *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_cond $e''+2;$
$\ulcorner e'+2, \mathcal{STMT}_2, e'' \urcorner$
$e''+1$ goto $e+1;$
$\ulcorner e''+2, \mathcal{STMT}_1, e \urcorner;$

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

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

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

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

(7){ *Assume that* $s = e'+1$ }
{ *as* $e'+1$ if_cond $e''+2$ *and* $e''+2$ *and* $e'+2$ *are labels in the compilation of the statement, we get a contradiction in this case* }

(8){ *Assume that* $s = e''+1$ }
{ *as* $e''+1$ goto $e+1$ *we get a contradiction once again*

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 :

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

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:*

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

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 $l = [i_1 : \texttt{instr} \ldots i_n : \texttt{instr}]$ in the compilation of method $\texttt{m}$is such that*

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

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

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 s : `instr` 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 .

this is not well explained

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.

alternative formulation: say that the only loop entries are the first instructions of a loop body

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* k : `instr` *which is a loop entry in the sense of definition* **??** *(i.e. there exists* j : `instr` *such that* j : `instr` $\rightarrow^l$ k : `instr` *) only if* $\mathcal{STMT}$ *is a loop or contains a substatement* $\mathcal{STMT}'$ *which is a loop statement and the following holds:*

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

$$\vee$$

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

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

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

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

target the instruction $e+1$: `instr` are in the same execution relation with $e+1$: `instr`,i.e. if $e+1$: `instr` 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* ($\exists k, s \leq k \leq e, k$: `instr` $\rightarrow^{l} e+1$: `instr`) $\iff$ ($\forall k, s \leq k \leq e, k$: `instr` $\rightarrow e+1$: `instr` $\Rightarrow k$: `instr` $\rightarrow^{l} e+1$: `instr`)

give the definition of the function *isExcHandlerStart*(k : `instr`)

Property 3.5.9 (Exception handler property for statements) *For every statement $\mathcal{STMT}$ which is not a try catch in method* `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:*

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

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

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

This means that there exists two elements $(s_1, e_1, eH_1, \texttt{Exc})$ and $(s_2, e_2, eH_2, \texttt{Exc})$ in the exception handler table of method `m` `m.excHndlS` such that :

$$\textit{(6)} \quad eH_1 \neq eH_2$$

From lemma 3.5.6 we get that :

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

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

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

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

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

which is in contradiction with *(6)*.

A similar property can be established about expressions

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:*

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

The 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` $(\mathit{ExcType}$ **var**$)\{\mathcal{STMT}_2\}$ *its compilation*

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

is such that the following holds

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

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** $+ -\ldots$) and **cntr** .

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

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

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

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

4.2 Weakest Predicate Transformer for the Source Language

The weakest precondition calculates for every statement $\mathcal{STMT}$ in method `m` from our source language, for any normal postcondition *Post* and exceptional postcondition function $\mathsf{excPost}^{src}$ ($\texttt{Exc} \rightarrow \mathcal{STMT} \rightarrow \mathcal{F}$), 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 **??** we discuss how the exceptional postcondition function is defined. Subsections **??** and **??** 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 `Exc` if it terminates on exception `Exc`

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

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

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 `Exc` is thrown.

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

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

4.2.2 Expressions

The weakest precondition function for expressions has the following signature:

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

For calculating the wp^{src} predicate of an expression $\mathcal{E}^{src}$ declared in method `m`, the function wp^{src} takes as arguments $\mathcal{E}^{src}$, a postcondition nPost^{src}, an exceptional postcondition function $\mathsf{excPost}^{src}$ and an $\mathcal{E}^{spec}$ and returns the formula $\mathrm{wp}^{src}(\mathcal{E}^{src}, \psi, \mathsf{excPost}^{src}, \texttt{m})_{\mathcal{E}^{spec}}$ 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
  ($const \in \{\textbf{constInt}, true, false, \textbf{constRef}\}$)

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

- field access expression

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

- arithmetic expressions

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

- method invokation