Mobius

Information Society Technologies

Project N°: **FP6-015905**

Project Acronym: **MOBIUS**

Project Title: **Mobility, Ubiquity and Security**

Instrument: **Integrated Project**

Priority 2: **Information Society Technologies**

Future and Emerging Technologies

Deliverable D4.3

Intermediate report on proof-transforming compiler

Due date of deliverable: 2007-09-01(T0+24)

Actual submission date: 2007-10-04

Start date of the project: **1 September 2005** Duration: **48 months**

Organisation name of lead contractor for this deliverable: **ETH**

Revision: Creation

Project co-funded by the European Commission in the Sixth Framework Programme (2002-2006)		
Dissemination level		
PU	Public	✓
PP	Restricted to other programme participants (including Commission Services)	
RE	Restricted to a group specified by the consortium (including Commission Services)	
CO	Confidential, only for members of the consortium (including Commission Services)	

Contributions

Site	Contributed to chapter
ETH	1, 3, 5, 8
INRIA	2, 4, 6, 7

Executive Summary
Intermediate report on proof-transforming compiler

This document provides an intermediate report on the work within Task 4.4 (Proof-Transforming Compiler) of the MOBIUS project, co-funded by the European Commission within the Sixth Framework Programme (FP6-015905). Full information on this project, including the contents of this deliverable, is available online at `http://mobius.inria.fr`.

PCC requires code producers to provide a certificate that their code has the desired properties. For simple program properties that are similar in complexity to type safety, classical PCC uses so-called certifying compilers to produce proofs automatically from source programs. For the advanced security properties considered in this project, additional techniques are necessary to produce certificates: First, source programs are analyzed by advanced type systems, and similar type systems are used on the bytecode level, where they allow efficient checking of security properties. Producing correctly typed bytecode programs from source code, requires a *type-preserving compiler* that translates the extended type information from source to bytecode programs. Second, whenever logic-based techniques are used to prove the desired properties, these techniques should be applied to source programs in order to benefit from the higher level of abstraction compared to byte code. We generalize the notion of certifying compilers to *proof-transforming compilers* that transform specifications and proofs of source programs to the bytecode level.

This deliverable reports on our efforts to develop type-preserving and proof-transforming compilers from Java to bytecode. In particular, it summarizes the following major contributions: (1) An architecture for proof-transforming compilers, which is the basis of future implementation work in this task. (2) An encoding of JML specifications in first order logic, which is needed to verify source programs. (3) A proof-transformation scheme for a subset of Java. This scheme preserves proof obligations during the translation. Therefore, proofs for these obligations on the source level can be re-used on the bytecode level. (4) A proof-transformation scheme for Java's abrupt termination features. This scheme handles the subtle interactions between finally-clauses and break-statements. (5) A type preserving compiler for the secure information flow type system developed in Task 2.1.

These results clearly show the feasibility of type-preserving compilation for advanced type system and of the novel concept of proof-transforming compilers.

Contents

1 Introduction **6**

1.1 Motivation . . . 6

1.2 Architecture of the proof-transforming compiler . . . 6

1.3 Outline of the deliverable . . . 8

2 Source and bytecode language **10**

2.1 Simple imperative languages . . . 10

2.1.1 Source language: $\text{JAVA}_{\mathcal{B}}$ . . . 10

2.1.2 Bytecode language: $\text{JVM}_{\mathcal{B}}$ . . . 10

2.1.3 Compilation . . . 11

2.2 Adding objects and methods . . . 14

2.2.1 Source language: $\text{JAVA}_{\mathcal{O}}$ . . . 14

2.2.2 Bytecode language: $\text{JVM}_{\mathcal{O}}$ . . . 16

2.2.3 Compilation . . . 16

2.3 Adding exceptions . . . 17

2.3.1 Source language: $\text{JAVA}_{\mathcal{E}}$ . . . 17

2.3.2 Bytecode language . . . 18

2.3.3 Compiler from $\text{JAVA}_{\mathcal{E}}$ to $\text{JVM}_{\mathcal{E}}$ . . . 18

3 Translation from JML to FOL specifications **25**

3.1 Overview . . . 25

3.1.1 Routine annotations . . . 25

3.1.2 Local annotations . . . 26

3.2 Translation of JML . . . 26

3.2.1 JML type specifications . . . 26

3.2.2 JML Routine specifications . . . 28

3.2.3 JML expressions . . . 30

3.2.4 JML statements . . . 30

4 Preservation of proof obligations using a VCGen **31**

4.1 Preservation of proof obligations for simple imperative programs . . . 31

4.1.1 Verification condition generator for $\text{JAVA}_{\mathcal{B}}$ . . . 31

4.1.2 The verification condition generator for $\text{JVM}_{\mathcal{B}}$ . . . 33

4.1.3 Relation between the verification calculi for $\text{JAVA}_{\mathcal{B}}$ and $\text{JVM}_{\mathcal{B}}$ . . . 36

4.2 Preservation of proof obligations for objects and methods . . . 37

4.2.1 Method specification and behavior subtyping . . . 38

4.2.2 Verification condition generator for $\text{JAVA}_{\mathcal{O}}$ . . . 39

4.2.3 Verification conditions generator for $\text{JVM}_{\mathcal{O}}$ . . . 42

4.2.4 Relation between the verification calculi for $\text{JAVA}_{\mathcal{O}}$ and $\text{JVM}_{\mathcal{O}}$ . . . 44

4.3 Preservation of proof obligations for exceptions . . . 44

4.3.1 Verification conditions for $\text{JAVA}_{\mathcal{E}}$ . . . 44

4.3.2 Verification conditions for $JVM_{\mathcal{E}}$ . . . 46
4.3.3 Relation between the verification calculi for $JAVA_{\mathcal{E}}$ and $JVM_{\mathcal{E}}$ . . . 46

5 Proof-transforming compilation for programs with abrupt termination 53
5.1 Source language and logic . . . 53
5.1.1 Method and statement specifications . . . 54
5.1.2 Rules . . . 54
5.2 Bytecode language and logic . . . 56
5.2.1 Method and Instruction Specifications . . . 56
5.2.2 Rules . . . 57
5.3 Proof Translation . . . 57
5.3.1 Compositional Statement . . . 58
5.3.2 While Statement . . . 59
5.3.3 Try-Finally Statement . . . 59
5.3.4 Break Statement . . . 61
5.4 Example . . . 62
5.5 Soundness Theorem . . . 63

6 Implementation issues 65
6.1 Architecture refinement . . . 65
6.1.1 Mobius PVE . . . 65
6.1.2 Concrete architecture of the direct VCGen . . . 66
6.2 Source Logic for the PTC . . . 67
6.2.1 Pre-processing . . . 67
6.2.2 Verification conditions generation . . . 68
6.2.3 A backend to Bicolano . . . 69

7 Type-preserving compilation for a type system for secure information flow 71
7.1 Control dependence regions . . . 71
7.2 High level security type system . . . 71
7.3 Intermediate type system for source code . . . 75
7.4 Connecting the high level and intermediate type systems . . . 79
7.5 Target language . . . 80
7.6 Compilation. . . . 82
7.7 Connecting the intermediate and target type systems . . . 83
7.8 Discussion . . . 84

8 Conclusions and future work 86
8.1 Proof-transforming compilation . . . 86
8.2 Type-preserving compilation . . . 87

Chapter 1

Introduction

This deliverable describes the current state of the work done in Task 4.4. We concentrate on results of ETH and INRIA on proof-transforming compilation for non-optimized bytecode and type-preserving compilation for an information-flow type system. IC and IoC have also started their work on type-preserving compilation and proof-transforming compilation for optimized bytecode, respectively. Their work will be reported in the second deliverable for this task.

1.1 Motivation

The core idea of Proof-Carrying Code [30] is to associate with executable code a proof that the execution of the code satisfies desirable properties such as type safety or memory safety. These proofs can then be checked efficiently and mechanically by the code consumer.

Certifying compilers [31, 11] allow code producers to generate a proof of safety properties automatically during the compilation of source programs. For instance, certifying compilers for Java use the memory safety of the source language to provide a certificate for the memory safety of the generated bytecode.

The goal of the MOBIUS project is to verify programs wrt. non-trivial security or functional properties. In general, certificates for such properties cannot be generated automatically; certain proof obligations have to be discarded interactively. This interactive verification step is easier on the source than on the byte code level for several reasons: (1) Source programs have structured control flow, whereas bytecode may contain arbitrary jumps, which complicate reasoning. (2) Source code provides complex expressions, whereas bytecode works on an operand stack. (3) The Java source language has a slightly richer type system than bytecode, for instance, it provides a type `boolean`, which is not present on the bytecode level and which makes proof obligations simpler. Besides logic-based verification, type-based reasoning and, in particular, combinations of both benefit from the higher abstraction level of source code.

We propose to reason about programs on the source level and to generate certificates for bytecode automatically from the corresponding certificates for source code. For type-based reasoning, this translation step is known as type-preserving compilation [25]. For logic-based verification, we propose the novel concept of proof-transforming compilation. In Task 4.4, we have developed a Proof-Transforming Compiler and a Type-Preserving Compiler. Combining these two techniques will allow us to automatically generate hybrid certificates for bytecode using type-based and logic-based reasoning on the source level.

1.2 Architecture of the proof-transforming compiler

In this section, we describe the architecture of the compiler developed in Task 4.4. A proof-transforming compiler is significantly more complex than a type-preserving compiler because it involves programs, specifications, and proofs. Therefore, we focus on proof-transforming compilation in this section. Type-preserving compilation will be discussed in Chapter 7.

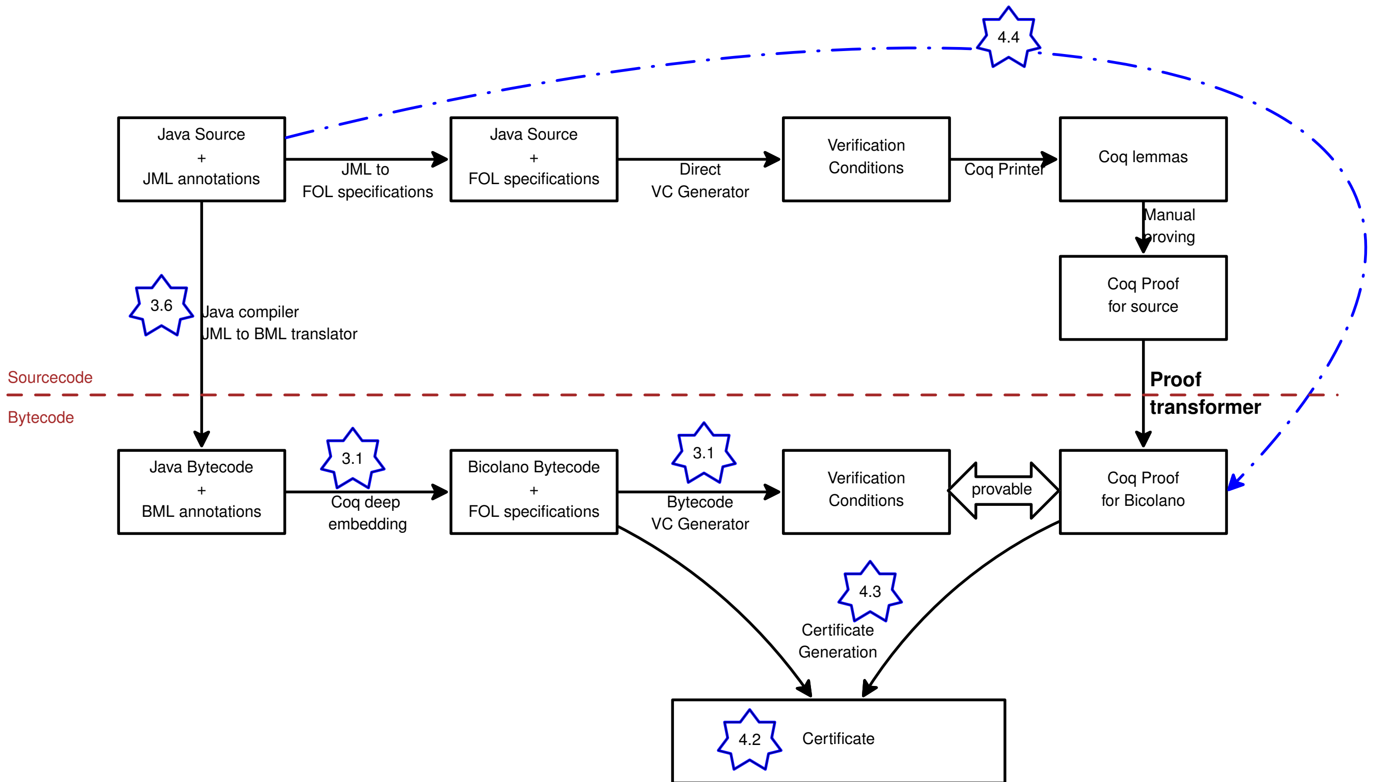


Figure 1.1: Overview of the proof transformation setting. We depict data by boxes and computations by arrows. We refer to tasks of the project using stars. The dashed arrow highlights the approach followed in this task.

Overview Figure 1.1 illustrates the scenarios for logic-based verification supported by the MOBIUS project. A proof for a bytecode program can be developed in two ways. (1) The bytecode and its BML specification are either developed directly or obtained from an ordinary Java compiler and a JML to BML translator. Using the technology developed in Task 3.1, we can then generate and prove the verification conditions for the bytecode program. The resulting proofs can be fed into the certificate generator. This process is depicted by the lower layer in Figure 1.1. (2) Alternatively, we can generate and prove verification conditions from the Java/JML source program and then translate the proofs using the proof-transforming compiler. This process is depicted by the upper layer in Figure 1.1. Task 4.4 focuses on the latter approach.

The architecture of a proof-transforming compiler is to a large extent determined by the structure and format of the source and bytecode verification conditions and proofs. In our case, the bytecode format is determined by the bytecode VCGen developed in Task 3.1. This direct VCGen takes Bicolano [32] with annotations in first order logic and generates first order logic verification conditions in Coq [23].

For source code, we developed a verification condition generator that leads to verification conditions and proofs similar to—or in the ideal case [8] the same as—the ones for bytecode. This choice makes the transformation of proofs feasible. Consequently, we did not build on an existing VCGen like the one used in Jack [19, 5] or ESC/Java2 [16, 10] because they produce structurally different verification conditions.

Trusted code base. A key consideration of proof-transforming compiler infrastructures is to keep the trusted code base small and simple. In our proof-transforming compiler architecture, only the underlying theorem prover Coq as well as the JVM model Bicolano are part of the trusted code base. Every other component of the framework is not part of the trusted code base for the following reasons:

- The bytecode VCGen which is used to check the certificate has been formally proved sound w.r.t. to

bicolano in Task 3.1.

- If one of the tools for producing source proofs (the upper layer in Figure 1.1) contains an error, invalid proofs may be generated. However, these invalid proofs will be rejected by the proof checker such that safety is not compromised. This is analogous to classical Proof-Carrying Code, where certifying compiler is also not part of the trusted code base.
- If the compilation process from JML annotated Java to BML annotated bytecode or the embedding of classfiles in Coq contains an error, a wrong program or wrong specifications are generated. Since the program and specifications are part of the certificate, code consumers detect such problems.

Dependencies Task 4.4 is using many results from other tasks and produces itself output that will be used in Task 5.4. The most important interplays are described below.

Task 2.1 The information flow type system developed in Task 2.1 is used as target type system for type preserving compilation shown in Chapter 7.

Task 3.1 The results of Task 3.1 are used at various places to realize a proof transforming compiler. The proof-transforming compiler produces a proof that can be applied to verification conditions that have been generated by the bytecode VCGen from Task 3.1. This is important as this VCGen is proved correct and does not have to be trusted. The bytecode specification language BML that was developed as part of Task 3.1 is needed in order to express the same properties on bytecode that can be expressed by JML on source code. We also use the relation that has been proved in Task 3.1 between the MOBIUS base logic and the Bannwart-Müller bytecode logic [4] that we used to show proof transformation for a Hoare logic.

Task 3.6 This tasks main goal is to produce the MOBIUS PVE which contains many tools that will seamlessly work together. As part of Task 3.6, a JML to BML translator will be built that is needed for the proof-transforming compiler. Also the deep embedding of classfiles in Coq is done by a Java tool called bico which is produced as part of Task 3.6.

Task 4.2 and 4.3 Our proof-transforming compiler generates proofs that can directly be used by the certificate generator defined in Task 4.3 to built certificates in the sense of Task 4.2.

At the end of the day, Task 4.4 provides a mean to use the technologies and tools developed by other tasks on bytecode level also on JML annotated source code.

1.3 Outline of the deliverable

Chapters 2 to 6 of this deliverable describe the steps of the verification process shown in Figure 1.1. Chapter 7 focuses on type-reserving compilation, and Chapter 8 concludes. We provide a more detailed outline in the following:

Chapter 2 describes the language subset that we take into consideration and provides an operational semantics for it.

Chapter 3 presents the first step of the verification generation for source code, namely the transformation of JML annotations into first order logic specifications. This transformation replaces JML specifications by a specification table. This table contains first order logic conditions associated with methods or individual Java statements.

Chapter 4 presents the direct VC generation as well as the key idea behind the proof transformer, which is the concept of preservation of proof obligations. In other words, this chapter shows that if we prove the verification conditions on source code, we can also prove the corresponding bytecode with the transformed proof.

Chapter 5 presents some highlights of proof transformation in presence of abrupt termination for a Hoare logic. Even though the proof-transforming compiler works with a verification condition generator, we have also investigated proof transformation for Hoare logics as it enables the transformation of proofs directly to the more powerful MOBIUS base logic.

Chapter 6 concentrates on the implementation issues of the VC generator. This chapter provides the interested reader with a more detailed architecture description and describes the integration of the source logic in ESC/Java2.

Chapter 7 presents a type-preserving compilation for the information flow type system developed in Task 2.1 [22].

Chapter 2

Source and bytecode language

We introduce the source language and bytecode language in three steps: First, we start with a simple imperative language, then we extend it with objects and method, and we conclude by adding exceptions. For each step, we present the source language and its semantics, the bytecode language, and the compilation scheme.

2.1 Simple imperative languages

In this section, we assume that a program consists of a single method.

2.1.1 Source language: $\text{JAVA}_{\mathcal{B}}$

Syntax Figure 2.1 defines the basic structured language $\text{JAVA}_{\mathcal{B}}$. Let $\mathcal{X}$ be the set of variables and $\mathcal{V}$ be the set of values ($\mathbb{Z}$ for the moment). $\mathcal{E}$ denotes the set of expressions and *stmt* the set of statements. In this language, a program $\mathcal{P}$ is simply an instruction followed by a return (i.e., $\mathcal{P} = stmt; \mathsf{return}\ e$)

Operational semantics The basic language does not deal with the heap. So, the language contains only a local memory $\rho : \mathcal{X} \to \mathcal{V}$, which is a mapping from local variables to values. We denote by $\mathcal{L}$ the set of local memories. Figure 2.2 gives the big step semantics of the basic source language. The first relation $\overset{\rho}{\hookrightarrow} \subseteq (\mathcal{E} \times \mathcal{L}) \times \mathcal{V}$ defines the evaluation of an expression e into a local memory ρ; the result is a value. Abusing notation, we use the same syntax for the evaluation of tests.

The set $\text{State}_{\mathcal{I}}$ of states is defined by the set of pairs $[i, \rho]$. The semantics of an instruction is defined, using a big step style, by the second relation $\Downarrow_{\mathcal{S}} \subseteq (\mathcal{I} \times \mathcal{L}) \times (\mathcal{L})$. This relation takes an instruction i in a local memory ρ, and returns the resulting local memory. There is no rule for the return statement, which can only appear at the end of the program. Finally, The evaluation a program $\mathcal{P} : \rho_0 \Downarrow_{\mathcal{S}} v$ is defined by:

$$\frac{\mathcal{P} = stmt; \mathsf{return}\ e \quad [\rho_0,\ stmt] \Downarrow_{\mathcal{S}} \rho \quad e \overset{\rho}{\hookrightarrow} v}{\mathcal{P} : \rho_0 \Downarrow_{\mathcal{S}} v}$$

2.1.2 Bytecode language: $\text{JVM}_{\mathcal{B}}$

Language A bytecode program $\dot{\mathcal{P}}$ is an array of bytecode instructions (defined figure 2.3). $\mathcal{P}_c$ is the set of program counters (index in the array of bytecode instructions). bytecode instructions perform actions on the operand stack (push and load push values on the stack, binop performs an operation with the two top elements, store saves the top element in a variable) or control the execution flow (goto for a unconditional jump and if for a conditional jump).

operations	op	::=	$+ \mid - \mid \times \mid /$	
comparisons	cmp	::=	$< \mid \leq \mid = \mid \neq \mid \geq \mid >$	
expressions	e	::=	$x \mid c \mid e$ op e	
			where $c \in \mathbb{Z}$ and $x \in \mathcal{X}$	
tests	t	::=	e cmp e	
statements	*stmt*	::=	$x := e$	assignment
		|	$\mathsf{if}(t)\{stmt\}\{stmt\}$	conditional
		|	$\mathsf{while}(t)\{stmt\}$	loop
		|	$stmt; stmt$	sequence
		|	skip	skip

Figure 2.1: Instruction set for the basic language JAVA$_{\mathcal{B}}$

$$\frac{}{x \overset{\rho}{\hookrightarrow} \rho(x)} \qquad \frac{}{c \overset{\rho}{\hookrightarrow} c} \qquad \frac{e_1 \overset{\rho}{\hookrightarrow} v_1 \quad e_2 \overset{\rho}{\hookrightarrow} v_2}{e_1 \text{ op } e_2 \overset{\rho}{\hookrightarrow} v_1 \text{ op } v_2} \qquad \frac{e_1 \overset{\rho}{\hookrightarrow} v_1 \quad e_2 \overset{\rho}{\hookrightarrow} v_2}{e_1 \text{ cmp } e_2 \overset{\rho}{\hookrightarrow} v_1 \text{ cmp } v_2}$$

$$\frac{e \overset{\rho}{\hookrightarrow} v}{[x := e,\, \rho] \Downarrow_{\mathcal{S}} \rho\{x \mapsto v\}} \qquad \frac{}{[\mathsf{skip},\, \rho] \Downarrow_{\mathcal{S}} \rho} \qquad \frac{[stmt_1,\, \rho] \Downarrow_{\mathcal{S}} \rho' \quad [\rho',\, stmt_2] \Downarrow_{\mathcal{S}} \rho''}{[stmt_1; stmt_2,\, \rho] \Downarrow_{\mathcal{S}} \rho''}$$

$$\frac{t \overset{\rho}{\hookrightarrow} \mathsf{true} \quad [i_t,\, \rho] \Downarrow_{\mathcal{S}} \rho'}{[\mathsf{if}(t)\{i_t\}\{i_f\},\, \rho] \Downarrow_{\mathcal{S}} \rho'} \qquad \frac{t \overset{\rho}{\hookrightarrow} \mathsf{false} \quad [i_f,\, \rho] \Downarrow_{\mathcal{S}} \rho'}{[\mathsf{if}(t)\{i_t\}\{i_f\},\, \rho] \Downarrow_{\mathcal{S}} \rho'}$$

$$\frac{t \overset{\rho}{\hookrightarrow} \mathsf{true} \quad [i,\, \rho] \Downarrow_{\mathcal{S}} \rho' \quad [\rho',\, \mathsf{while}(t)\{i\}] \Downarrow_{\mathcal{S}} \rho''}{[\mathsf{while}(t)\{i\},\, \rho] \Downarrow_{\mathcal{S}} \rho''} \qquad \frac{t \overset{\rho}{\hookrightarrow} \mathsf{false}}{[\mathsf{while}(t)\{i\},\, \rho] \Downarrow_{\mathcal{S}} \rho}$$

Figure 2.2: Semantics of the basic language

Bytecode Semantics An abstract machine state is a triple $\langle k,\, \rho,\, os\rangle$, where k is the program counter, ρ a mapping from variables to values, and os the operand stack. For the moment, the operand stack only contains intermediate values needed by the evaluation of source language expressions. We denote by $\text{State}_{\mathcal{B}}$ the set of abstract machine states. The small step semantics of a bytecode program is given by the relation $\rightsquigarrow \subseteq \text{State}_{\mathcal{B}} \times (\text{State}_{\mathcal{B}} + \mathcal{V})$, which represents one step of execution (defined in figure 2.4). The transitive closure of $\rightsquigarrow$ to a final value is inductively defined by:

$$\frac{\langle k,\, \rho,\, os\rangle \rightsquigarrow v}{\langle k,\, \rho,\, os\rangle \Downarrow v} \qquad \frac{\langle k,\, \rho,\, os\rangle \rightsquigarrow \langle k',\, \rho',\, os'\rangle \quad \langle k',\, \rho',\, os'\rangle \Downarrow v}{\langle k,\, \rho,\, os\rangle \Downarrow v}$$

Finally, the evaluation of a bytecode program $\dot{\mathcal{P}} : \rho \Downarrow v$, from a initial mapping of local variables to a final value is defined by

$$\dot{\mathcal{P}} : \rho \Downarrow v \overset{\text{def}}{\equiv} \langle 0,\, \rho,\, \emptyset\rangle \Downarrow v$$

2.1.3 Compilation

The compiler used here is defined in figure 2.5 by two functions. The first one compiles the expressions $[\![e]\!]$ and generates a bytecode sequence which evaluates e and stores/pushes the result on the top of the operand stack. The second one compiles the instructions $k : [\![i]\!]$. The argument k indicates the starting position

intructions	i	$::=$	$\mathsf{push}\ c$	push value on top of stack
		$\mid$	binop op	binary operation on stack
		$\mid$	$\mathsf{load}\ x$	load value of x on stack
		$\mid$	$\mathsf{store}\ x$	store top of stack in variable x
		$\mid$	$\mathsf{goto}\ j$	unconditional jump
		$\mid$	if cmp j	conditional jump
		$\mid$	return	return the top value of the stack

where $c \in \mathbb{Z}$, $x \in \mathcal{X}$, and $j \in \mathcal{P}_c$.

Figure 2.3: Instruction set for the basic bytecode

$$\frac{\dot{\mathcal{P}}[k] = \mathsf{push}\ c}{\langle k,\, \rho,\, os\rangle \rightsquigarrow \langle k+1,\, \rho,\, c :: os\rangle} \qquad \frac{\dot{\mathcal{P}}[k] = \mathsf{binop}\ \mathrm{op} \quad v = v_1\ \mathrm{op}\ v_2}{\langle k,\, \rho,\, v_1 :: v_2 :: os\rangle \rightsquigarrow \langle k+1,\, \rho,\, v :: os\rangle} \qquad \frac{\dot{\mathcal{P}}[k] = \mathsf{return}}{\langle k,\, \rho,\, v :: os\rangle \rightsquigarrow v}$$

$$\frac{\dot{\mathcal{P}}[k] = \mathsf{load}\ x}{\langle k,\, \rho,\, os\rangle \rightsquigarrow \langle k+1,\, \rho,\, \rho(x) :: os\rangle} \qquad \frac{\dot{\mathcal{P}}[k] = \mathsf{store}\ x}{\langle k,\, \rho,\, v :: os\rangle \rightsquigarrow \langle k+1,\, \rho\{x \mapsto v\},\, os\rangle}$$

$$\frac{\dot{\mathcal{P}}[k] = \mathsf{goto}\ j}{\langle k,\, \rho,\, os\rangle \rightsquigarrow \langle j,\, \rho,\, os\rangle} \qquad \frac{\dot{\mathcal{P}}[k] = \mathsf{if}\ \mathrm{cmp}\ j \quad v_1\ \mathrm{cmp}\ v_2 = \mathsf{true}}{\langle k,\, \rho,\, v_1 :: v_2 :: os\rangle \rightsquigarrow \langle j,\, \rho,\, os\rangle} \qquad \frac{\dot{\mathcal{P}}[k] = \mathsf{if}\ \mathrm{cmp}\ j \quad v_1\ \mathrm{cmp}\ v_2 = \mathsf{false}}{\langle k,\, \rho,\, v_1 :: v_2 :: os\rangle \rightsquigarrow \langle k+1,\, \rho,\, os\rangle}$$

Figure 2.4: Semantics of the basic bytecode

of the resulting bytecode sequence in the final bytecode program; this information is used to compute the labels attached to branching instructions. Abusing notation, we will write $k : [\![e]\!]$ for the compilation of an expression starting at position k.

Compilation of an assignment $x := e$ is the compilation of the expression e followed by $\mathsf{store}\ x$. At the end of the evaluation of $[\![e]\!]$, the value of e is on the top of the operand stack, then a $\mathsf{store}\ x$ instruction stores this value in the variable x and pops the value from the stack.

The compilation of a conditional $k : [\![\mathsf{if}(e_1\ op\ e_2)\{stmt_1\}\{stmt_2\}]\!]$ starts by the sequence corresponding to the evaluation of the two expressions e_2 and e_1. After this sequence the operand stack contains on the top the values of e_1 and e_2. The $\mathsf{if}\ cmp\ k_2$ instruction evaluates the comparison and pops the two values from the stack. If the test is true, the evaluation continues at label k_1 corresponding to the beginning of the true branch; otherwise, the if instruction jumps to label k_2 corresponding to the beginning of the false branch. At the end of the false branch a goto instruction jumps the code of the false branch.

The compilation of a loop $k : [\![\mathsf{while}(e_1\ cmp\ e_2)\{i\}]\!]$ evaluates the two expressions e_2 and e_1 and then performs a conditional jump. If the test is false the evaluation jumps to the code corresponding to the body of the loop, if the test is true the evaluation continues by the evaluation of the loop body and then performs a jump to the label corresponding to the beginning of the evaluation of the test.

Finally, the compilation of a program $\mathcal{P} = (i; \mathsf{return}\ e)$ is defined by:

$$[\![\mathcal{P}]\!] \stackrel{\text{def}}{\equiv} 0 : [\![i]\!];\ [\![e]\!];\ \mathsf{return}$$

Correctness of the compiler The correctness of a compiler expresses that the semantics of a source program is the same as of the compiled version. In other words, for all source programs $\mathcal{P}$, if $\mathcal{P} : \rho_0 \Downarrow_{\mathcal{S}} v$ then the evaluation of its compiled version $\dot{\mathcal{P}}$ starting form the initial memory ρ_0 leads to the same resulting

Compilation of expressions

$$
\begin{array}{rcl}
k:[\![x]\!] & = & k : \mathsf{load}\ x \\
k:[\![c]\!] & = & k : \mathsf{push}\ c \\
k:[\![e_1\ op\ e_2]\!] & = & k : [\![e_1]\!];\ k_1 : [\![e_2]\!];\ k_2 : \mathsf{binop}\ op \\
\text{where } k_1 & = & k + |[\![e_1]\!]| \\
k_2 & = & k_1 + |[\![e_2]\!]|
\end{array}
$$

Compilation of statements

$$
\begin{array}{rcl}
k:[\![x := e]\!] & = & k : [\![e]\!];\ k_1 :\ \mathsf{store}\ x \\
\text{where } k_1 & = & k + |[\![e]\!]| \\
 & & \\
k:[\![stmt_1; stmt_2]\!] & = & k:[\![stmt_1]\!]; k_2:[\![stmt_2]\!] \\
\text{where } k_2 & = & k + |[\![stmt_1]\!]| \\
 & & \\
k:[\![\mathsf{return}\ e]\!] & = & [\![e]\!];\ \mathsf{return} \\
 & & \\
k:[\![\mathsf{if}(e_1\ cmp\ e_2)\{stmt_1\}\{stmt_2\}]\!] & = & k : [\![e_2]\!]; k_1\ [\![e_1]\!]; k_2 :\ \mathsf{if}\ cmp\ k_3; \\
 & & k_2 + 1:[\![stmt_1]\!];\ k_3 : \mathsf{goto}\ l;\ k_3 + 1:[\![stmt_2]\!] \\
\text{where } k_1 & = & k + |[\![e_2]\!]| \\
k_2 & = & k_1 + |[\![e_1]\!]| \\
k_3 & = & k_2 + |[\![stmt_1]\!]| + 1 \\
l & = & k_3 + |[\![stmt_2]\!]| + 1 \\
 & & \\
k:[\![\mathsf{while}(e_1\ cmp\ e_2)\{stmt\}]\!] & = & k : \mathsf{goto}\ k_1; k + 1:[\![stmt]\!]; \\
 & & k_1 : [\![e_2]\!];\ k_2 : [\![e_1]\!];\ k_3 : \mathsf{if}\ cmp\ k + 1; \\
\text{where } k_1 & = & k + 1 + |[\![stmt]\!]| \\
k_2 & = & k_1 + |[\![e_2]\!]| \\
k_3 & = & k_2 + |[\![e_1]\!]|
\end{array}
$$

Figure 2.5: COMPILATION SCHEME

value v. The correctness proof of a compiler is done by exhibiting a simulation between the evaluation of the source program and the evaluation of the bytecode program.

Lemma 1 (Correctness for expressions) *For all* bytecode *programs $\dot{\mathcal{P}}$, expressions e, values v, memories ρ, and operand stacks os such that $l = |[\![e]\!]|$ and $\dot{\mathcal{P}}[k..k + l] = [\![e]\!]$ the following property holds:*

$$e \overset{\rho}{\hookrightarrow} v \Rightarrow \langle k,\ \rho,\ os\rangle \rightsquigarrow^* \langle k + l,\ \rho,\ v :: os\rangle$$

Lemma 2 (Correctness for instructions) *For all* bytecode *programs $\dot{\mathcal{P}}$, instructions i, memories ρ and ρ' such that $l = |[\![i]\!]|$ and $\dot{\mathcal{P}}[k..k + l] = k:[\![i]\!]$ the following property holds:*

$$[i,\ \rho] \Downarrow_{\mathcal{S}} \rho' \Rightarrow \langle k,\ \rho,\ \emptyset\rangle \rightsquigarrow^* \langle k + l,\ \rho',\ \emptyset\rangle$$

Lemma 3 (Correctness of the compiler) *For all source programs $\mathcal{P}$, if $\mathcal{P} : \rho_0 \Downarrow_{\mathcal{S}} \rho, v$ then its compiled version evaluates to the same result:*

$$\mathcal{P} : \rho_0 \Downarrow_{\mathcal{S}} v \Rightarrow [\![\mathcal{P}]\!] : \rho_0 \Downarrow v$$

2.2 Adding objects and methods

Now that we have introduced the basic concepts of the languages and of the compilation scheme, we shall consider several more realistic extensions of the language. In this section particularly, we shall focus on a language with object-oriented features.

2.2.1 Source language: $\text{JAVA}_{\mathcal{O}}$

$\text{JAVA}_{\mathcal{O}}$ is an extension of the simple language $\text{JAVA}_{\mathcal{B}}$ for dealing with objects and methods.

Syntax In this setting, a program is a set of classes taken from the set $\mathcal{C}$, which is partially ordered $(\mathcal{C}, \preceq)$. The hierarchy of classes will be used to resolve virtual calls. Classes are entities that are provided with a list of methods taken from the set $\mathcal{M}$. A method is identified uniquely by its identifier. Classes are also provided with a list of fields taken from the set $\mathcal{F}$.

The new syntactic constructs that $\text{JAVA}_{\mathcal{O}}$ provides in addition to $\text{JAVA}_{\mathcal{B}}$ are:

$$\begin{array}{lrcl} \text{expressions} & e & ::= & \ldots \mid \mathsf{this} \mid e.\mathsf{m}(e) \mid e.\mathsf{f} \mid \mathsf{new}\ \mathsf{C} \\ \text{statements} & \mathit{stmt} & ::= & \ldots \mid e.\mathsf{f} := e \end{array}$$

The expression construct $e.\mathsf{m}(e_1)$ denotes the invocation of method with identifier m on the receiver e and the argument e_1. For simplicity, we restrict ourselves to method with one argument, the extension to the general case being straightforward. The construct $\mathsf{new}\ \mathsf{C}$ denotes a creation of a new instance of type C. We also provide the language with object field access expressions $e.\mathsf{f}$, where f is from the set of field names $\mathcal{F}$. Moreover, we allow the update of object fields via the instruction $e.\mathsf{f} := e$. The expression this is a special variable referring to the current object, i.e., the object of the method currently being executed.

Global memory Compared to $\text{JAVA}_{\mathcal{B}}$, the set of $\text{JAVA}_{\mathcal{O}}$ values is extended to $\mathcal{V} = \mathbb{Z} \cup \mathcal{R}$, where $\mathcal{R}$ is an (infinite) set of references. Note that for the moment, we ignore the issue of dereferencing a variable which does not contain a proper reference value (i.e., null pointers). We extend the notion of program state to include a global memory (heap), which will give meaning to object expressions. Thus, a state is the pair $(\rho,\ h)$ of the local memory store ρ and the global object store h.

The notion of a heap is defined axiomatically similarly to Poetzsch-Heffter and Müller's work [33]. We define an abstract data type $\mathcal{H}$ of heap objects and define the following operations over it:

$$\begin{array}{ll} \mathsf{Dom} & : \mathcal{H} \times \mathcal{R} \to \mathit{bool} \\ \mathsf{Get} & : \mathcal{H} \times \mathcal{R} \times \mathcal{F} \to \mathcal{V} \\ \mathsf{Update} & : \mathcal{H} \times \mathcal{R} \times \mathcal{F} \times \mathcal{V} \to \mathcal{H} \\ \mathsf{New} & : \mathcal{H} \times \mathcal{C} \to \mathcal{H} \times \mathcal{R} \\ \mathsf{typeof} & : \mathcal{R} \to \mathcal{C} \end{array}$$

The function $\mathsf{Get}(h, r, \mathsf{f})$ returns the value stored in the field f of object reference r in heap h. Usually this operator returns the value stored in the field f (if it exists) of any allocated reference r; otherwise, it is undefined. Since the type checking of Java can ensure this properties, we assume that the function is total. We define Dom such that $\mathsf{Dom}(h, r)$ holds if r is allocated in the heap h otherwise it returns false. The function $\mathsf{Update}(h, r, \mathsf{f}, v)$ updates the value stored in the field f of object location r in heap h. The function $\mathsf{New}(h, \mathsf{C})$ returns a pair of a fresh reference and the resulting heap augmented with the new reference. The function typeof maps references to reference types. For simplicity, we use the notation $h(r.\mathsf{f})$ for $\mathsf{Get}(h, r, \mathsf{f})$ and $h\{r.\mathsf{f} \mapsto v\}$ for $\mathsf{Update}(h, r, \mathsf{f}, v)$.

We then assume a set of properties to hold for those functions. For illustration, we require that if a field of a particular reference is updated with a new value, then an access to the field for this reference will evaluate to that new value:

$$h\{r.\mathsf{f} \mapsto v\}(r.\mathsf{f}) = v \tag{2.1}$$

When allocating a new object in a heap, then the resulting heap has the same domain plus the new object:

$$\mathsf{New}(h, \mathsf{C}) = (h', r) \Rightarrow (\forall l, \mathsf{Dom}(h, l) \Rightarrow \mathsf{Dom}(h', l)) \wedge \mathsf{Dom}(h', r) \wedge \neg\mathsf{Dom}(h, r) \wedge \mathsf{typeof}(r) = \mathsf{C} \qquad (2.2)$$

For the full set of axioms, the reader may refer to [33].

Semantics The first difference with the semantics of JAVA$_\mathcal{B}$ is that expressions can now have side effects (on the heap) due to the allocation of new objects or to method invocations. The consequence is that the evaluation of expressions now returns a value (the result of the expression) and the resulting heap. Note that the local memory, containing the value of local variables, still remains unchanged. A state is now a triple $[i, \rho, h] \in \mathcal{I} \times \mathcal{L} \times \mathcal{H}$.

The semantics of JAVA$_\mathcal{O}$i given Fig. 2.6. There is now three mutually defined predicates:

- $(e, h) \overset{\rho}{\hookrightarrow} (v, h')$ stand for: the evaluation of the expression e in the heap h and local memory ρ leads to the value v and the heap h'
- $[m, r, v, h] \Downarrow_\mathcal{S} (v, h')$ stand for: the evaluation of the method m in receiver r and argument v in the heap h leads to the value v and the heap h'
- $[i, \rho, h] \Downarrow_\mathcal{S} (\rho', h')$ stand for: the evaluation of the instruction i in the local memory ρ and the heap h leads to the local memory ρ' and the heap h'

Compared to JAVA$_\mathcal{O}$ the evaluation of some expressions may cause side effects as for instance method invocation expressions or instance creation. The new rules for field access and instance creation are straightforward.

The semantics of method calls is more complicated because in Java methods are dispatched dynamically. This means that the actual method to be executed cannot be determined statically. In languages like Java, this is due to method overriding in subclasses.

Definition 2.2.1 (Method overriding)
A method m_2 *declared in class* C_2 *is said to override a method* m_1 *declared in class* C_1 *and we denote by* $\mathsf{overrides}(\mathsf{m}_2, \mathsf{m}_1)$ *the following properties:*

- *the class* C_1 *is an ancestor class of* C_2*, denoted by* $(\mathsf{C}_2 \preceq \mathsf{C}_1)$
- *methods* m_1 *and* m_2 *have the same signature (the same short name and type)*

An important property of the lookup function is that for all m m' such that $\mathsf{m}' = \mathsf{lookup}(\mathsf{m}, C)$ then $\mathsf{overrides}(\mathsf{m}', \mathsf{m})$.

To model dynamic dispatching, we assume that there is a function lookup_P attached to each program P that takes a method identifier m and a class name and returns the identifier of the method to be executed. Once the method m to be executed is determined, the function $\mathsf{body}(\mathsf{m})$ returns the method body of m which corresponds to a program statement JAVA$_\mathcal{B}$. The method body of the latter is then executed with the resulting heap of the evaluation of the argument and the receiver.

The evaluation of a method is defined in the same way as the evaluation of a JAVA$_\mathcal{B}$ program, the only difference is that the local memory is initialized with the value of the receiver for the variable this and the value of the argument for the variable *arg*.

Compare to JAVA$_\mathcal{B}$ the evaluation of the instructions is modified to take into account the side effect of the expressions.

2.2.2 Bytecode language: $\mathrm{JVM}_{\mathcal{O}}$

A $\mathrm{JVM}_{\mathcal{O}}$ program is the same that a $\mathrm{JAVA}_{\mathcal{O}}$ program, except that the body function associate to a method an array of instructions. To simplify the presentation we use the notation $P_m[k]$ as a short cut for $\mathsf{body}(m)[k]$.

The set of instructions of $\mathrm{JVM}_{\mathcal{B}}$ is extended with additional instructions to create new objects, to read or update fields objects, and invoke m to call virtual methods. Here, m is a method identifier, which may correspond to several methods in the class hierarchy due to overriding of methods.

intructions	i	::=	push c	push value on top of stack
		|	binop op	binary operation on stack
		|	load x	load value of x on stack
		|	store x	store top of stack in variable x
		|	goto j	unconditional jump
		|	if cmp j	conditional jump
		|	return	return the top value of the stack
		|	new C	instance creation
		|	getfield f	field access
		|	putfield f	field assignment
		|	invoke m	virtual method call

As for $\mathrm{JAVA}_{\mathcal{O}}$, we assume that there is a function lookup_P attached to each program P, which takes a method and a class name and returns the method to be executed.

Operational semantics

While JVM states contain a frame stack to handle method invocations, it is convenient for showing the correctness of the verification condition generator to rely on an equivalent semantics where method invocation is performed in one big step transition. Hence, a $\mathrm{JVM}_{\mathcal{O}}$ state is of the form $\langle k, \rho, os, h\rangle$, where k, ρ, and os are defined as in $\mathrm{JVM}_{\mathcal{B}}$ and h is a heap. We use an intermediate semantics - small step for all instructions except for method invocations and big step for method invocations; more precisely, the semantics directly calls the full evaluation of the called method from an initial state to a return value and uses it to continue the current computation. As we can see from the rules of the operational semantics, the instructions have a similar but not exactly the same semantics as their counterparts in $\mathrm{JAVA}_{\mathcal{O}}$. The differences come from the specific features of the virtual machine as for instance the operand stack used for expression evaluation. Also, the bytecode language is more fine grained—the composition of several bytecode instructions will correspond to a $\mathrm{JAVA}_{\mathcal{O}}$ expression.

The operational semantics, given in Figure 2.7, is defined by two mutually dependent predicates. The instruction putfield f updates field f with the value on top of the operand stack, for the receiver stored second to top on the stack; after the execution of the instruction, the two top stack elements are removed. The instruction getfield f pushes the value of the field f for the object reference at the top of the operand stack.

2.2.3 Compilation

The main difference between a $\mathrm{JAVA}_{\mathcal{O}}$ and a $\mathrm{JVM}_{\mathcal{O}}$ program is in the representation of methods. The structures of the programs are the same except that the body of a method in the source is a source instruction following by a return whereas it is an array of bytecode instruction for the target language.

So, the compiler from $\mathrm{JAVA}_{\mathcal{O}}$ to $\mathrm{JVM}_{\mathcal{O}}$ is a mapping translating each method of the source program to an array of bytecode instructions. The compilation scheme for instruction is essentially the same as the one for $\mathrm{JAVA}_{\mathcal{B}}$. We simply give the new rules for the new constructs in Fig. 2.8. The correctness of the compiler can be proved in the same way as for $\mathrm{JAVA}_{\mathcal{B}}$. The different lemmas are modified to include the global memory.

Lemma 4 (Correctness of the JAVA$_\mathcal{O}$ compiler) *For all* JAVA$_\mathcal{O}$ *program* P *and is compiled version* $\dot{\mathcal{P}}$ *and for all method* m *in* P *(and so* $\dot{\mathcal{P}}$*), if* $[\mathsf{m}, r, v, h_0] \Downarrow_{\mathcal{S}} (v, h)$ *then* $\langle 1, \{\mathsf{this} \mapsto r, arg \mapsto v\}, \emptyset, h_0\rangle \Downarrow_{\mathsf{m}} (v, h)$

2.3 Adding exceptions

In the previous section, we looked at objects and method calls where the semantics allowed only for one kind of termination, namely normal termination. We actually worked under the assumption that executions always go well, for instance we assumed that all variables are initialized with a proper value before any use. In practice, programming languages do not work under such hypothesis and must somehow handle the situation where variables which does not have proper values are used. An example for such situation is the access of a field of a location via a non-initialized variable. Actually, programming languages use an exception mechanism to cope with these cases. In the following, we shall describe the extensions of the language which provide such a mechanism.

2.3.1 Source language: JAVA$_\mathcal{E}$

First, we extend the set of possible values $\mathcal{V}$ with the new value null, which stands for the reference that does not point into the heap; that is, $\mathcal{V} = \mathbb{Z} \cup \mathcal{R} \cup \mathsf{null}$. Fig. 2.9 gives the new semantics of expressions, which may either terminate normally or exceptionally. An exception is simply an object of an exception type. In this setting, the final state of an expression evaluation is either the normal state configuration (v, h) in case the evaluation yields the value v for the expression and leaves the execution with a heap h, or the exceptional state $(\mathsf{exc}\ r, h)$ where r is a reference of an exception type and h is the resulting heap. The execution of statement terminates either in a normal state (ρ, h) with a memory ρ and heap h respectively, or in an exceptional state of the form $(\mathsf{exc}\ r, \rho, h)$ where r is the thrown exception object, ρ is the current memory and h is the current heap.

There are five rules for the evaluation of method invocations. First, the receiver object is evaluated and then the argument. If one of these evaluations leads to an exception, the exception is propagated. If both evaluate normally then if the value corresponding to the receiver object is null, the method invocation terminates abruptly with a NullPntrExc exception. Otherwise, the rule is the same as for JAVA$_\mathcal{O}$, except if the method body terminates exceptionally, then the evaluation of the method call also terminates exceptionally.

Field access expression evaluation has two possible executions: if the dereferenced field evaluates to a reference different from null then it behaves normally as in the previous section; if the dereferenced field is null then exception NullPntrExc is returned. For brevity, we do not give the other contextual rules, which propagate exceptional termination of expression.

Moreover, the language supports a mechanism for handling thrown exceptions. We extend our language with the following statement instructions:

$$\text{statements} \quad stmt \quad ::= \quad \ldots \mid \mathsf{try}\{stmt\}\ \mathsf{catch}\ (\mathsf{EType})\ \{stmt\} \mid \mathsf{throw}\ e \mid$$

The semantics of the extended instructions is given in Fig. 2.10. In our setting, the terminal states of statement executions may be either normal or exceptional. The normal terminal state is of the form (ρ, h) and indicates that statement execution leaves the memory ρ and the heap h. The exceptional execution is of the form $(\mathsf{exc}\ r, \rho, h)$ and shows that the statement has terminated by throwing the exception r and the local memory is now ρ and the heap is h. The semantics of the try catch statement is such that exceptions thrown in the try statement of type or subtype of EType will be handled by the catch statement, and the whole statement terminates execution as the catch statement; otherwise, if the exception r thrown by the try statement is not a type or subtype of EType, then the whole statement terminates exceptionally and leaves the execution exceptionally with the exception value EType and the resulting heap. The throw e statement allow programmers to throw exceptions explicitly. Notice that its execution always terminates exceptionally. If the exception expression is not null and then e evaluates to the exceptional value r, the execution of the throw statement terminates exceptionally with the exception r. Otherwise, if e is null, then the statement

terminates with a NullPntrExc exception. Similarly, the execution of field update statement has a normal and exceptional termination. As for expressions, we do not give all the rules for the propagation of exceptions.

2.3.2 Bytecode language

In this section, we consider an extension $\text{JVM}_{\mathcal{E}}$ of the $\text{JVM}_{\mathcal{O}}$ with an exception handling mechanism. Programs are similar to those in the $\text{JVM}_{\mathcal{O}}$ model. However, the instruction set of the $\text{JVM}_{\mathcal{O}}$ is extended with the bytecode throw.

Furthermore, we assume that programs come equipped with a partial function[1] $\mathsf{Handler}_{\mathsf{m}} : \mathcal{P}_c \times \mathcal{C} \rightharpoonup \mathcal{P}_c$ that for each method m selects the appropriate handler for a given program point. If an exception of class $C \in \mathcal{C}$ is thrown at program point $k \in \mathcal{P}_c$ then, if $\mathsf{Handler}_{\mathsf{m}}(k, C) = l$, the control will be transferred to program point l, and if $\mathsf{Handler}_{\mathsf{m}}(k, C)$ is undefined (noted $\mathsf{Handler}_{\mathsf{m}}(k, C) \uparrow$), the exception is uncaught in method m.

Operational semantics

We give in Figure 2.11 the semantics of exception-throwing instructions in $\text{JVM}_{\mathcal{E}}$. Rules for the rest of the instructions are as in $\text{JVM}_{\mathcal{O}}$. The exception throwing instructions now have several rules, which correspond to the normal and exceptional termination of the instruction. Moreover, there are separate rules for the exceptional termination when the exception is caught or not.

For instance, there are three more rules for the virtual call instruction. The first is for normal execution. The second is for the case of the called method terminates abruptly but the exception is handle. The third express the case where the received object is null, and the NullPntrExc is caught. The two last rules, correspond to the cases where the evaluation of the call leads to an uncaught exception. We use NullPntrExc as the class associated to the null pointer exception. Each instruction which performs an access on a reference (getfield f, putfield f and throw) have a similar semantics.

2.3.3 Compiler from $\text{JAVA}_{\mathcal{E}}$ to $\text{JVM}_{\mathcal{E}}$

In the following, we shall extend the compiler function presented in Section 2.2.3 to deal with the new statements concerning exception handling and throwing. The compilation of the statements considered in the previous sections for the basic and object oriented language are the same. That is why we omit them here and show in Fig 2.12 the compilation of the new language constructs.

Notice that the compiler constructs the exception handler function Handler simultaneously with the compilation of $\text{JAVA}_{\mathcal{E}}$ statements to $\text{JVM}_{\mathcal{E}}$ instructions. To simplify the presentation the exception handler function is built incrementally (using imperative style). The only construct which effects the exception handler function is the try catch statement. First the instruction $stmt_1$ (the try statement) is compiled, after that the exception handler have been extended in such way that if an exception is throw during the execution of $stmt_1$ then the handler catch it if the exception has the appropriate type, which leads to the execution of the catch statement $stmt_2$. If no exception is thrown during the execution of $stmt_1$ the instruction goto k_2 will jump the code of $stmt_2$ to continue normally the execution of the program.

The compilation of the throw consists in the compilation of the expression e followed by the instruction throw and it does not change the exception handler function.

Lemma 5 (Correctness of the compiler) *A $\text{JAVA}_{\mathcal{E}}$ program P and its compiled version $\dot{\mathcal{P}}$ share the same semantics.*

[1]This opaque handling function hides the notions of handler list and subclass used in Java.

$$\frac{}{(x,h) \overset{\rho}{\hookrightarrow} (\rho(x),h)} \qquad \frac{}{(c,h) \overset{\rho}{\hookrightarrow} (c,h)}$$

$$\frac{(e_2,h) \overset{\rho}{\hookrightarrow} (v_2,h_2) \quad (e_1,h_2) \overset{\rho}{\hookrightarrow} (v_1,h_1)}{(e_1 \text{ op } e_2,h) \overset{\rho}{\hookrightarrow} (v_1 \text{ op } v_2,h_1)} \qquad \frac{(e_2,h) \overset{\rho}{\hookrightarrow} (v_2,h_2) \quad (e_1,h_2) \overset{\rho}{\hookrightarrow} (v_1,h_1)}{(e_1 \text{ cmp } e_2,h) \overset{\rho}{\hookrightarrow} (v_1 \text{ cmp } v_2,h_1)}$$

$$\frac{(e,h) \overset{\rho}{\hookrightarrow} (r,h')}{(e.\mathsf{f},h) \overset{\rho}{\hookrightarrow} (h'(r.\mathsf{f}),h')} \qquad \frac{\mathsf{New}(h,\mathsf{C}) = (h',r)}{(\mathsf{new}\ \mathsf{C},h) \overset{\rho}{\hookrightarrow} (r,\ h')}$$

$$\frac{\begin{array}{c}(e,h) \overset{\rho}{\hookrightarrow} (r,\ h_1) \quad (e_1,h_1) \overset{\rho}{\hookrightarrow} (v_1,\ h_2) \\ \mathsf{m}' = \mathsf{lookup}_P(\mathsf{m},\mathsf{typeof}(r)) \quad [\mathsf{m}',r,v_1,h_2] \Downarrow_{\mathcal{S}} (v,\ h')\end{array}}{(e.\mathsf{m}(e_1),h) \overset{\rho}{\hookrightarrow} (v,\ h')}$$

$$\frac{\mathsf{body}(\mathsf{m}) = i;\ \mathsf{return}\ e \quad [i,\{\mathsf{this} \mapsto r, \mathit{arg} \mapsto v\},h] \Downarrow_{\mathcal{S}} (\rho',h') \quad (e,h') \overset{\rho'}{\hookrightarrow} (v',h'')}{[\mathsf{m},r,v,h] \Downarrow_{\mathcal{S}} (v',h'')}$$

$$\frac{(e,h) \overset{\rho}{\hookrightarrow} (v,h')}{[x := e,\rho,h] \Downarrow_{\mathcal{S}} (\rho\{x \mapsto v\},h')} \qquad \frac{}{[\mathsf{skip},\rho,h] \Downarrow_{\mathcal{S}} (\rho,\mathsf{h})}$$

$$\frac{[\mathit{stmt}_1,\rho,h] \Downarrow_{\mathcal{S}} (\rho_1,h_1) \quad [\mathit{stmt}_2,\rho_2,h_1] \Downarrow_{\mathcal{S}} (\rho_2,h_2)}{[\mathit{stmt}_1;\mathit{stmt}_2,\rho,h] \Downarrow_{\mathcal{S}} (\rho_2,h_2)}$$

$$\frac{(t,h) \overset{\rho}{\hookrightarrow} (\mathsf{true},h_1) \quad [i_t,\rho,h_1] \Downarrow_{\mathcal{S}} (\rho_2,h_2)}{[\mathsf{if}(t)\{i_t\}\{i_f\},\rho,h] \Downarrow_{\mathcal{S}} (\rho_2,h_2)} \qquad \frac{(t,h) \overset{\rho}{\hookrightarrow} (\mathsf{false},h_1) \quad [i_f,\rho,h_1] \Downarrow_{\mathcal{S}} (\rho_2,h_2)}{[\mathsf{if}(t)\{i_t\}\{i_f\},\rho,h] \Downarrow_{\mathcal{S}} (\rho_2,h_2)}$$

$$\frac{(t,h) \overset{\rho}{\hookrightarrow} (\mathsf{true},h_1) \quad [i,\rho,h_1] \Downarrow_{\mathcal{S}} (\rho_2,h_2) \quad [\mathsf{while}(t)\{i\},\rho_2,h_2] \Downarrow_{\mathcal{S}} (\rho_3,h_3)}{[\mathsf{while}(t)\{i\},\rho,h] \Downarrow_{\mathcal{S}} (\rho_3,h_3)}$$

$$\frac{(t,h) \overset{\rho}{\hookrightarrow} (\mathsf{false},h')}{[\mathsf{while}(t)\{i\},\rho,h] \Downarrow_{\mathcal{S}} (\rho,h')}$$

$$\frac{(e_1,h) \overset{\rho}{\hookrightarrow} (r,\ h_1) \quad (e_2,h_1) \overset{\rho}{\hookrightarrow} (v_2,\ h_2)}{[e_1.\mathsf{f} := e_2,\rho,h] \Downarrow_{\mathcal{S}} (\rho,\ h_2\{r.\mathsf{f} \mapsto v_2\})}$$

Figure 2.6: SEMANTICS OF JAVA$_{\mathcal{O}}$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{push}\ c}{\langle k, \rho, os, h\rangle \overset{\mathsf{m}}{\leadsto} \langle k+1, \rho, c :: os, h\rangle} \qquad \frac{P_{\mathsf{m}}[k] = \mathsf{binop}\ \mathrm{op} \quad v = v_1\ \mathrm{op}\ v_2}{\langle k, \rho, os, h\rangle \overset{\mathsf{m}}{\leadsto} \langle k+1, \rho, v :: os, h\rangle}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{load}\ x}{\langle k, \rho, os, h\rangle \overset{\mathsf{m}}{\leadsto} \langle k+1, \rho, \rho(x) :: os, h\rangle} \qquad \frac{P_{\mathsf{m}}[k] = \mathsf{store}\ x}{\langle k, \rho, v :: os, h\rangle \overset{\mathsf{m}}{\leadsto} \langle k+1, \rho\{x \mapsto v\}, os, h\rangle}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{goto}\ j}{\langle k, \rho, os, h\rangle \overset{\mathsf{m}}{\leadsto} \langle j, \rho, os, h\rangle}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{if}\ \mathrm{cmp}\ j \quad v_1\ \mathrm{cmp}\ v_2 = \mathsf{true}}{\langle k, \rho, v_1 :: v_2 :: os, h\rangle \overset{\mathsf{m}}{\leadsto} \langle k+1, \rho, os, h\rangle} \qquad \frac{P_{\mathsf{m}}[k] = \mathsf{if}\ \mathrm{cmp}\ j \quad v_1\ \mathrm{cmp}\ v_2 = \mathsf{false}}{\langle k, \rho, v_1 :: v_2 :: os, h\rangle \overset{\mathsf{m}}{\leadsto} \langle j, \rho, os, h\rangle}$$

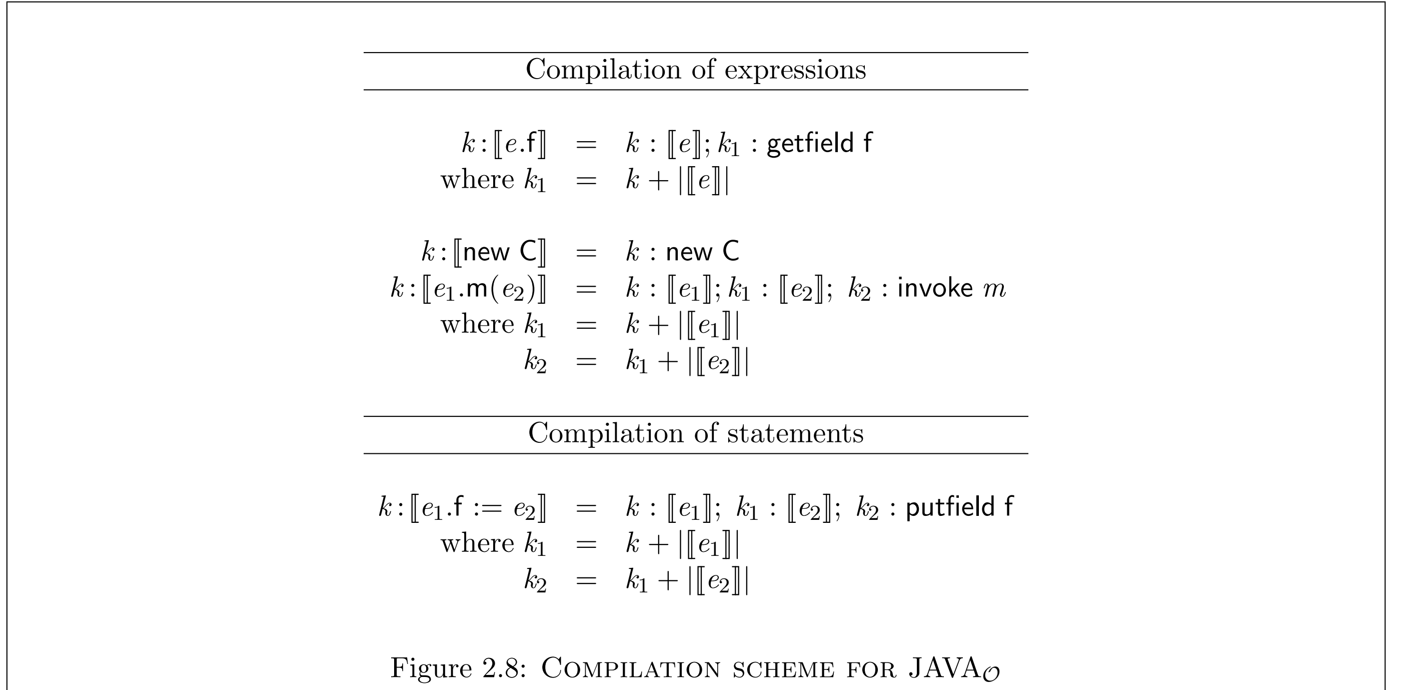
$$\frac{P_{\mathsf{m}}[k] = \mathsf{new}\ C \qquad \mathsf{New}(h, \mathsf{C}) = (r, h')}{\langle k, \rho, os, h\rangle \overset{\mathsf{m}}{\leadsto} \langle k+1, \rho, r :: os, h'\rangle}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{getfield}\ \mathsf{f}}{\langle k, \rho, r :: os, h\rangle \overset{\mathsf{m}}{\leadsto} \langle k+1, \rho, h(r.\mathsf{f}) :: os, h\rangle} \qquad \frac{P_{\mathsf{m}}[k] = \mathsf{putfield}\ \mathsf{f}}{\langle k, \rho, v :: r :: os, h\rangle \overset{\mathsf{m}}{\leadsto} \langle k+1, \rho, os, h\{r.\mathsf{f} \mapsto v\}\rangle}$$

$$\frac{\begin{array}{c} P_{\mathsf{m}}[k] = \mathsf{invoke}\ \mathsf{m}_1 \qquad \mathsf{m}' = \mathsf{lookup}_P(\mathsf{m}_1, \mathsf{typeof}(r)) \\ \langle 1, \{\mathsf{this} \mapsto r, arg \mapsto v\}, \epsilon, h\rangle \Downarrow_{\mathsf{m}'} (v, h') \end{array}}{\langle k, \rho, v :: r :: os, h\rangle \overset{\mathsf{m}}{\leadsto} \langle k+1, \rho, v :: os, h'\rangle}$$

$$\frac{S \overset{\mathsf{m}}{\leadsto} S' \quad S' \Downarrow_{\mathsf{m}} (v, h)}{S \Downarrow_{\mathsf{m}} (v, h)} \qquad \frac{P_{\mathsf{m}}[k] = \mathsf{return}}{\langle k, \rho, v :: os, h\rangle \Downarrow_{\mathsf{m}} (v, h)}$$

Figure 2.7: Operational Semantics for $\mathrm{JVM}_{\mathcal{O}}$



Compilation of expressions

$$
\begin{array}{rcl}
k:[\![e.\mathsf{f}]\!] & = & k:[\![e]\!]; k_1 : \mathsf{getfield\ f} \\
\text{where } k_1 & = & k + |[\![e]\!]| \\
\\
k:[\![\mathsf{new\ C}]\!] & = & k : \mathsf{new\ C} \\
k:[\![e_1.\mathsf{m}(e_2)]\!] & = & k:[\![e_1]\!]; k_1 : [\![e_2]\!];\ k_2 : \mathsf{invoke}\ m \\
\text{where } k_1 & = & k + |[\![e_1]\!]| \\
k_2 & = & k_1 + |[\![e_2]\!]|
\end{array}
$$

Compilation of statements

$$
\begin{array}{rcl}
k:[\![e_1.\mathsf{f} := e_2]\!] & = & k:[\![e_1]\!];\ k_1 : [\![e_2]\!];\ k_2 : \mathsf{putfield\ f} \\
\text{where } k_1 & = & k + |[\![e_1]\!]| \\
k_2 & = & k_1 + |[\![e_2]\!]|
\end{array}
$$

Figure 2.8: Compilation scheme for JAVA$_{\mathcal{O}}$

$$
\frac{(e,h) \xhookrightarrow{\rho} (\mathsf{exc}\ r,\ h_1)}{(e.\mathsf{m}(e_1),h) \xhookrightarrow{\rho} (\mathsf{exc}\ r,\ h_1)}
\qquad
\frac{(e,h) \xhookrightarrow{\rho} (v,\ h_1) \qquad (e_1,h_1) \xhookrightarrow{\rho} (\mathsf{exc}\ r,\ h_2)}{(e.\mathsf{m}(e_1),h) \xhookrightarrow{\rho} (\mathsf{exc}\ r,\ h_2)}
$$

$$
\frac{(e,h) \xhookrightarrow{\rho} (\mathsf{null},\ h') \qquad (e_1,h') \xhookrightarrow{\rho} (v_1,\ h_1) \qquad \mathsf{New}(h_1,\mathsf{NullPntrExc}) = (\mathbf{np},h_2)}{(e.\mathsf{m}(e_1),h) \xhookrightarrow{\rho} (\mathsf{exc}\ \mathbf{np},\ h_2)}
$$

$$
\frac{\begin{array}{c}(e,h) \xhookrightarrow{\rho} (r_1,\ h_1) \qquad (e_1,h_1) \xhookrightarrow{\rho} (v_1,\ h_2) \qquad \mathsf{m}' = \mathsf{lookup}_P(\mathsf{m},\mathsf{stypeof}(r)) \\ r_1 \neq \mathsf{null} \quad [\mathsf{m}',r_1,v_1,h_2] \Downarrow_{\mathcal{S}} (v,\ h')\end{array}}{(e.\mathsf{m}(e_1),h) \xhookrightarrow{\rho} (v,\ h')}
$$

$$
\frac{\begin{array}{c}(e,h) \xhookrightarrow{\rho} (r_1,\ h_1) \qquad (e_1,h_1) \xhookrightarrow{\rho} (v_1,\ h_2) \qquad \mathsf{m}' = \mathsf{lookup}_P(\mathsf{m},\mathsf{typeof}(r_1)) \\ r_1 \neq \mathsf{null} \quad [\mathsf{m}',r_1,v_1,h_2] \Downarrow_{\mathcal{S}} (\mathsf{exc}\ r,\ h')\end{array}}{(e.\mathsf{m}(e_1),h) \xhookrightarrow{\rho} (\mathsf{exc}\ r,\ h')}
$$

$$
\frac{(e,h) \xhookrightarrow{\rho} (v,\ h_1) \quad v \neq \mathsf{null}}{e.\mathsf{f} \xhookrightarrow{\rho} (h_1(v.\mathsf{f}),\ h_1)}
$$

$$
\frac{(e,h) \xhookrightarrow{\rho} (\mathsf{null},\ h') \quad \mathsf{New}(h',\mathsf{NullPntrExc}) = (\mathbf{np},h_1)}{(e.\mathsf{f},h) \xhookrightarrow{\rho} (\mathsf{exc}\ \mathbf{np},\ h_1)}
$$

Figure 2.9: Semantics of expression evaluation in JAVA$_{\mathcal{E}}$

$$\frac{(e_1, h) \stackrel{\rho}{\hookrightarrow} (\mathsf{exc}\ r,\ h_1)}{[e_1.\mathsf{f} := e_2,\ \rho,\ h] \Downarrow_{\mathcal{E}} (\mathsf{exc}\ r,\ \rho,\ h_1)} \qquad \frac{(e_1, h) \stackrel{\rho}{\hookrightarrow} (v_1,\ h_1) \qquad (e_2, h_1) \stackrel{\rho}{\hookrightarrow} (\mathsf{exc}\ r,\ h_2)}{[e_1.\mathsf{f} := e_2,\ \rho,\ h] \Downarrow_{\mathcal{E}} (\mathsf{exc}\ r,\ \rho,\ h_2)}$$

$$\frac{(e_1, h) \stackrel{\rho}{\hookrightarrow} (\mathsf{null},\ h_1) \qquad (e_2, h_1) \stackrel{\rho}{\hookrightarrow} (v_2,\ h_2) \qquad \mathsf{New}(h_2, \mathsf{NullPntrExc}) = (\mathbf{np}, h_3)}{[e_1.\mathsf{f} := e_2,\ \rho,\ h] \Downarrow_{\mathcal{E}} (\mathsf{exc}\ \mathbf{np},\ \rho,\ h_3)}$$

$$\frac{(e_1, h) \stackrel{\rho}{\hookrightarrow} (v_1,\ h_1) \quad e_2 \stackrel{\rho}{\hookrightarrow} h_1(v_2,\ h_2)}{[e_1.\mathsf{f} := e_2,\ \rho,\ h] \Downarrow_{\mathcal{E}} (\rho,\ h_2\{v_1.\mathsf{f} \mapsto v_2\})}$$

$$\frac{[stmt_1,\ \rho,\ h] \Downarrow_{\mathcal{E}} (\rho_1,\ h_1)}{[\mathsf{try}\{stmt_1\}\ \mathsf{catch}\ (\mathsf{EType})\ \{stmt_2\},\ \rho,\ h] \Downarrow_{\mathcal{E}} (\rho_1,\ h_1)}$$

$$\frac{[stmt_1,\ \rho,\ h] \Downarrow_{\mathcal{E}} (\mathsf{exc}\ r,\ \rho_1,\ h_1) \quad \neg\mathsf{typeof}(r) \preceq \mathsf{EType}}{[\mathsf{try}\{stmt_1\}\ \mathsf{catch}\ (\mathsf{EType})\ \{stmt_2\},\ \rho,\ h] \Downarrow_{\mathcal{E}} (\mathsf{exc}\ r,\ \rho_1,\ h_1)}$$

$$\frac{[stmt_1,\ \rho,\ h] \Downarrow_{\mathcal{E}} (\mathsf{exc}\ r,\ \rho_1,\ h_1) \quad \mathsf{typeof}(r) \preceq \mathsf{EType} \quad [stmt_2,\ \rho_1,\ h_1] \Downarrow_{\mathcal{E}} S}{[\mathsf{try}\{stmt_1\}\ \mathsf{catch}\ (\mathsf{EType})\ \{stmt_2\},\ \rho,\ h] \Downarrow_{\mathcal{E}} S}$$

$$\frac{(e, h) \stackrel{\rho}{\hookrightarrow} (r,\ h_1)}{[\mathsf{throw}\ e,\ \rho,\ h] \Downarrow_{\mathcal{E}} (\mathsf{exc}\ r,\ \rho,\ h_1)} \qquad \frac{(e, h) \stackrel{\rho}{\hookrightarrow} (\mathsf{null},\ h') \quad \mathsf{New}(h', \mathsf{NullPntrExc}) = (\mathbf{np}, h_1)}{[\mathsf{throw}\ e,\ \rho,\ h] \Downarrow_{\mathcal{E}} (\mathsf{exc}\ \mathbf{np},\ \rho,\ h_1)}$$

Figure 2.10: SEMANTICS OF STATEMENT EXECUTION IN $\mathrm{JAVA}_{\mathcal{E}}$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{getfield}\ \mathsf{f} \quad r \neq \mathsf{null}}{\langle k, \rho, r :: os, h\rangle \overset{\mathsf{m}}{\rightsquigarrow} \langle k+1, \rho, h(r).\mathsf{f} :: os, h\rangle}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{getfield}\ \mathsf{f} \quad \mathsf{New}(h, \mathsf{NullPntrExc}) = (\mathbf{np}, h_1) \quad \mathsf{Handler}_{\mathsf{m}}(k, \mathsf{NullPntrExc}) = j}{\langle k, \rho, \mathsf{null} :: os, h\rangle \overset{\mathsf{m}}{\rightsquigarrow} \langle j, \rho, \mathbf{np} :: \mathsf{nil}, h_1\rangle}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{putfield}\ \mathsf{f} \quad r \neq \mathsf{null}}{\langle k, \rho, v :: r :: os, h\rangle \overset{\mathsf{m}}{\rightsquigarrow} \langle k+1, \rho, os, h\{r.\mathsf{f} \mapsto v\}\rangle}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{putfield}\ \mathsf{f} \quad \mathsf{Handler}_{\mathsf{m}}(k, \mathsf{NullPntrExc}) = j \quad \mathsf{New}(h, \mathsf{NullPntrExc}) = (\mathbf{np}, h_1)}{\langle k, \rho, v :: \mathsf{null} :: os, h\rangle \overset{\mathsf{m}}{\rightsquigarrow} \langle j, \rho, \mathbf{np} :: \mathsf{nil}, h_1\rangle}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{throw} \quad v \neq \mathsf{null} \quad \mathsf{Handler}_{\mathsf{m}}(k, \mathsf{typeof}(v)) = j}{\langle k, \rho, v :: os, h\rangle \overset{\mathsf{m}}{\rightsquigarrow} \langle j, \rho, v :: \mathsf{nil}, h\rangle}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{throw} \quad \mathsf{Handler}_{\mathsf{m}}(k, \mathsf{NullPntrExc}) = j \quad \mathsf{New}(h, \mathsf{NullPntrExc}) = (\mathbf{np}, h_1)}{\langle k, \rho, \mathsf{null} :: os, h\rangle \overset{\mathsf{m}}{\rightsquigarrow} \langle j, \rho, \mathbf{np} :: \mathsf{nil}, h_1\rangle}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{invoke}\ \mathsf{m}_1 \quad \mathsf{m}' = \mathsf{lookup}_P(\mathsf{m}_1, \mathsf{typeof}(r)) \quad r \neq \mathsf{null} \quad \langle 1, \{\mathit{this} \mapsto r, \mathit{arg} \mapsto v_1\}, \mathsf{nil}, h\rangle \Downarrow_{\mathsf{m}'} \langle v, h'\rangle}{\langle k, \rho, v_1 :: r :: os, h\rangle \overset{\mathsf{m}}{\rightsquigarrow} \langle k+1, \rho, v :: os, h'\rangle}$$

$$\frac{\begin{array}{c} P_{\mathsf{m}}[k] = \mathsf{invoke}\ \mathsf{m}_1 \quad \mathsf{m}' = \mathsf{lookup}_P(\mathsf{m}_1, \mathsf{typeof}(r)) \quad r \neq \mathsf{null} \\ \langle 1, \{\mathit{this} \mapsto r, \mathit{arg} \mapsto v_1\}, \mathsf{nil}, h\rangle \Downarrow_{\mathsf{m}'} \langle \mathsf{exc}\ r', h'\rangle \quad \mathsf{Handler}_{\mathsf{m}}(k, \mathsf{typeof}(r')) = j \end{array}}{\langle k, \rho, v_1 :: r :: os, h\rangle \overset{\mathsf{m}}{\rightsquigarrow} \langle j, \rho, r' :: \mathsf{nil}, h'\rangle}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{invoke}\ \mathsf{m}_1 \quad \mathsf{Handler}_{\mathsf{m}}(k, \mathsf{NullPntrExc}) = j \quad \mathsf{New}(h, \mathsf{NullPntrExc}) = (\mathbf{np}, h_1)}{\langle k, \rho, v_1 :: \mathsf{null} :: os, h\rangle \overset{\mathsf{m}}{\rightsquigarrow} \langle j, \rho, \mathbf{np} :: \mathsf{nil}, h_1\rangle}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{getfield}\ \mathsf{f} \quad \mathsf{New}(h, \mathsf{NullPntrExc}) = (\mathbf{np}, h_1) \quad \mathsf{Handler}_{\mathsf{m}}(k, \mathsf{NullPntrExc}) \uparrow}{\langle k, \rho, \mathsf{null} :: os, h\rangle \Downarrow_{\mathsf{m}} \langle \mathsf{exc}\ \mathbf{np}, h_1\rangle}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{putfield}\ \mathsf{f} \quad \mathsf{Handler}_{\mathsf{m}}(k, \mathsf{NullPntrExc}) \uparrow \quad \mathsf{New}(h, \mathsf{NullPntrExc}) = (\mathbf{np}, h_1)}{\langle k, \rho, v :: \mathsf{null} :: os, h\rangle \Downarrow_{\mathsf{m}} \langle \mathsf{exc}\ \mathbf{np}, h_1\rangle}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{throw} \quad v \neq \mathsf{null} \quad \mathsf{Handler}_{\mathsf{m}}(k, \mathsf{typeof}(v)) \uparrow}{\langle k, \rho, v :: os, h\rangle \Downarrow_{\mathsf{m}} \langle \mathsf{exc}\ v, h\rangle}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{throw} \quad \mathsf{Handler}_{\mathsf{m}}(k, \mathsf{NullPntrExc}) \uparrow \quad \mathsf{New}(h, \mathsf{NullPntrExc}) = (\mathbf{np}, h_1)}{\langle k, \rho, \mathsf{null} :: os, h\rangle \Downarrow_{\mathsf{m}} \langle \mathsf{exc}\ \mathbf{np}, h_1\rangle}$$

$$\frac{\begin{array}{c} P_{\mathsf{m}}[k] = \mathsf{invoke}\ \mathsf{m}_1 \quad \mathsf{m}' = \mathsf{lookup}_P(\mathsf{m}_1, \mathsf{typeof}(r)) \quad r \neq \mathsf{null} \\ \langle 1, \{\mathit{this} \mapsto r, \mathit{arg} \mapsto v_1\}, \mathsf{nil}, h\rangle \Downarrow_{\mathsf{m}'} \langle \mathsf{exc}\ r', h'\rangle \quad \mathsf{Handler}_{\mathsf{m}}(k, \mathsf{typeof}(r')) \uparrow \end{array}}{\langle k, \rho, v_1 :: r :: os, h\rangle \Downarrow_{\mathsf{m}} \langle \mathsf{exc}\ r', h'\rangle}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{invoke}\ \mathsf{m}_1 \quad \mathsf{Handler}_{\mathsf{m}}(k, \mathsf{NullPntrExc}) \uparrow \quad \mathsf{New}(h, \mathsf{NullPntrExc}) = (\mathbf{np}, h_1)}{\langle k, \rho, v_1 :: \mathsf{null} :: os, h\rangle \Downarrow_{\mathsf{m}} \langle \mathsf{exc}\ \mathbf{np}, h_1\rangle}$$

Figure 2.11: OPERATIONAL SEMANTICS FOR $\mathrm{JVM}_{\mathcal{E}}$

$$
\begin{array}{rcl}
k : [\![\mathsf{throw}\ e]\!] & = & k : [\![e]\!];\ k_1 : \mathsf{throw} \\
\text{where } k_1 & = & k + |[\![e]\!]| \\
 & & \\
k : [\![\mathsf{try}\{stmt_1\}\ \mathsf{catch}\ (\mathsf{EType})\ \{stmt_2\}]\!] & = & k : [\![stmt_1]\!]; \mathsf{goto}\ \mathsf{k}_2; k_1 : [\![stmt_2]\!] \\
\text{where } k_1 & = & k + |[\![stmt_1]\!]| + 1 \\
k_2 & = & k_1 + |[\![stmt_2]\!]| \\
\mathsf{Handler}(j, E) & := & \begin{cases} k_1 & k \le j < k_1 \ \wedge\ E \preceq \mathsf{EType} \\ \mathsf{Handler}(j, E) & \text{otherwise} \end{cases}
\end{array}
$$

Figure 2.12: Compiling the exception handling constructs from $\mathrm{JAVA}_{\mathcal{E}}$ to $\mathrm{JVM}_{\mathcal{E}}$

Chapter 3

Translation from JML to FOL specifications

This chapter presents the translation of JML into first order logic (FOL) specifications. As illustrated in Figure 1.1, this translation is the first step of the Mobius Direct VCGen.

The translation covers most of the JML level 0 language. We omit the instance keyword (because the Java subset considered in this deliverable does not have interfaces), the keywords \nonnullelements and \elemtype (because they are just shortcuts for other specifications), model fields (because they would be a distraction), and the ownership modifiers peer and rep (because they can be encoded using constructs we support [14]).

Our translation formalizes essentially the semantics described in the JML reference manual [21]; we discuss deviations below. In the following, we present the overall approach of the translation and then discuss the translation of individual JML constructs. We focus on the essential ideas here. Details are presented in a technical report [12].

3.1 Overview

Our translation takes as input a Java program and its JML specification. It produces a specification table containing routine (method and constructor) annotations as well as local annotations (assertions) for individual Java statements. The predicates in the table and the local annotations are expressed in FOL. Both the specification table and the local annotations are fed into the verification condition generator described in Chapter 4.

In the following, we use the notation $Tr(e)$ to denote the FOL term that results from the translation of a JML annotation e.

3.1.1 Routine annotations

The specification table for routines annotations has the following form:

$$\Gamma(\mathcal{P}_e,\, \mathsf{C},\, \mathsf{m},\, \Phi,\, \Psi,\, \Psi_{exc})$$

For a routine m in class C of program $\mathcal{P}_e$, the specification table contains the precondition Φ, the normal postcondition Ψ, and several exceptional postconditions Ψ_{exc}, where exc denotes the thrown exception. Ψ and Ψ_{exc} contain a special variable for the return value and the exception object, respectively. We denote this variables by res and esc.

The FOL predicates in the specification table contain all the information that is available for the corresponding state, originating from various JML annotations such as requires, ensures, and signals clauses as well as invariant, assignable, and initially clauses. This means that the VCGen that deals with the predicates in the specification table does not have to deal with the large number of JML constructs.

Table 3.1 gives an overview on what routine annotations are influenced by the translation of the just mentioned JML constructs. Section 3.2 below describes the details of how the routine annotations are built from this JML constructs.

Precondition Φ	Postcondition Ψ	Exceptional Postconditions Ψ_{exc}
requires	ensures	signals
invariant	initially	signals_only
	assignable	initially
	invariant	assignable
		invariant

Table 3.1: The routine annotations together with the influencing JML constructs

3.1.2 Local annotations

Local annotations characterize the state before the execution of a particular Java statement. Typical examples are JML's assert statement and loop invariants. We encode them by associating each Java statement with zero or more local annotations of the following forms. Their interpretation in the VCGen is explained in Section 6.2. For a JML expression e we have:

assert $Tr(e)$**:** Condition $Tr(e)$ acts as a local assertion for the following Java-statement. It has to be proved to hold.

assume $Tr(e)$**:** In the VCGen, $Tr(e)$ is assumed at this point. This is only sound if $Tr(e)$ is known to hold, for instance, because the property was established by a type system or static analysis. It acts as an additional information for proving the current goal.

loop invariant $Tr(\mathcal{I})$**:** This annotation contains the translation of a loop invariant $\mathcal{I}$ that holds before and after each iteration of a loop.

set $Tr(\mathcal{V} = e)$**:** This annotation differs from the previous ones. It is used to translate ghost variable declarations and assignments (that is, JML's set command). It contains the name of the ghost variable and the FOL translation of the expression e, which is assigned to the ghost variable. This is done in the VCGen by substituting $\mathcal{V}$ by e.

3.2 Translation of JML

In this section, we present the translation of type specifications and routine specifications, JML expressions, and JML statements. For the translation, we use the variable heap to denote the current heap and _pre_heap to denote the heap in the prestate of the method.

3.2.1 JML type specifications

Object invariants

Object invariants are denoted by the JML clause invariant $\mathcal{I}$. To express $Tr(\mathcal{I})$, we introduce the predicate inv : $\mathcal{C} \times \mathcal{R} \to bool$, which yields for a given type and object its invariant.

In JML, all invariants have to hold in the prestates and poststates of routines that are not declared as helper (the so-called *visible states*). More concretely, non-helper methods can rely on the fact that in their prestate all allocated objects satisfy their invariant, and they have to ensure that the same property holds in their poststate. In order to be able to verify a system in the presence of possible re-entrancy (call-backs), it's also necessary to establish all invariants before calling a non-helper method.

To simplify verification considerably, we currently restrict the expressiveness of invariants. An invariant is only allowed to depend on fields that are defined in the class declaring the invariant. Fields inherited from superclasses must not be mentioned in invariants (see [28] for a detailed discussion). Our implementation will check the admissibility of invariants during the translation process.

For non-helper routines, invariants are conjoined to pre- and postconditions as follows:

- To the precondition Φ, we conjoin the term

  $$\forall o, t : \mathsf{Dom}(\mathsf{heap}, o) \wedge \mathsf{typeof}(o) \preceq t \Rightarrow \mathsf{inv}(t, o)$$

  where $\mathsf{typeof}(o) \preceq t$ yields whether the dynamic type of object o is a (not necessarily proper) subtype of type t.

  This precondtion expresses that the method can rely on the fact that all invariants have to hold in its prestate. Callers of the method have to live up to this precondition. The treatment of constructors is analogous, but the this object is excluded since its invariant has not yet been established.

- to the postconditions Ψ and Ψ_{exc}, we conjoin the term

  $$\forall o, t : \mathsf{Dom}(\mathsf{heap}, o) \wedge \mathsf{typeof}(o) \preceq t \wedge t \in \mathsf{modified}(m) \Rightarrow \mathsf{inv}(t, o)$$

  where $\mathsf{modified}(m)$ is the set of types which can be modified during execution of m.

  These postconditions express that the method has to satisfy the invariants of all objects that might have been modified. This is determined using a static analysis which makes an overestimation by collecting all types that can be assigned to. It is sufficient to check those invariants because their subtypes must not contain invariants that depend on inherited fields [28]. So in case of both normal and exceptional termination, we want to enforce the method to establish invariants that could have been broken during their execution.

The helper modifier can be applied to private routines with the effect that they do not depend on invariants. For helper routines, the above pre- and postconditions are not generated.

The initially-clause

The JML clause initially $\mathcal{P}$ expresses that every non-helper constructor of the enclosing class as well as all subclasses has to establish $\mathcal{P}$. The translation therefore conjoins $Tr(\mathcal{P})$ to Ψ and Ψ_{exc} for all affected constructors.

The modifier 'ghost'

The ghost modifier allows one to declare fields and local variables that only appear in JML specifications. For instance, ghost variables are used to carry resource information or to save an old value of a non-ghost variable. ghost fields and variables can be assigned to by the set statement and read within specifications in the same way as normal fields and variables.

Currently, we support only ghost variables, but an extension to ghost fields is straightforward. For each declaration of a ghost variable, a set annotation is created. If the declaration includes an expression for the initial value of the variable, it is also included in the set annotation, which is then attached to the following Java statement.

During the translation, we also use ghost variables to capture the prestate values of the arguments such that we can refer to them in \old expressions.

3.2.2 JML Routine specifications

Light- and heavyweight specifications

We desugar arbitrary routine specifications to behavior specification cases as follows:

Lightweight specification cases: JML does not define default values for omitted clauses in lightweight specifications. We are using the same default that are defined for heavyweight specifications except for the case of the signals clause which is handled stricter. The idea behind choosing the defaults like this is to be on the conservative side. If a routine is not specified, we have to assume that we loose all information about the program that we had before.

- The weakest possible precondition (true) is chosen if no requires clause is given. This means that no assumptions about the program state at the beginning of the method can be done to prove the method body.
- The weakest possible postcondition (true) is chosen if no ensures clause is given. Here, the caller of the method with true as postcondition does not get any information from the postcondition about the program state after the call.
- An omitted signals_only clause means that no other exception than the ones mentioned in the throws clause of a method can be thrown.
- A default signals clause is introduced if none was specified. It contains false for java.lang.Exception meaning that the method is by default not allowed to throw any exception.
- If no modifies clause is present, the weakest possible modifies clause is given which is '\everything'. This means that the method might possibly modify every allocated heap location.

Normal behavior specification cases: As in the lightweight specification case without a signals clause, we want to guarantee that no exception can be thrown. We introduce a signals clause that contains false for java.lang.Exception.

Exceptional behavior specification cases: We introduce ensures false in order to guarantee that in this case, the method is not supposed to terminate normally.

Besides this, we can transform several specification cases to one single case by rewriting the routine level annotations as follows:

requires: Several requires clauses in the same specification case are conjoined to one single requires clause as all of them have to be assumed at the beginning of a method. requires clauses in different specification cases are being disjuncted.

ensures: For a (already conjoined) requires clause $\mathcal{P}$ of a specification case and a current class C, the ensures clauses $\mathcal{Q}$ of the same specification case are transformed to

$$\backslash\mathsf{type}(\mathsf{this}) \preceq C \wedge \backslash\mathsf{old}(\mathcal{P}) \Rightarrow \mathcal{Q}$$

By doing this, the ensures clauses are being guarded by the precondition of the same specification case.

signals, signals_only: The same principle that we use in ensures clauses is used for exceptional postconditions. We guard the condition by the precondition of the same specification case.

assignable: We merge all assignable heap locations form several specification cases together.

In the following, the description of the translation for routine annotations is given for the desugared form.

The requires-clause

A requires $\mathcal{P}$ clause becomes part of the precondition. We conjoin $Tr(\mathcal{P})$ to Φ.

The ensures-clause

An ensures $\mathcal{P}$ clause becomes part of the normal postcondition. We conjoin $Tr(\mathcal{P})$ to Ψ.

The signals- and signals_only-clause

For the JML clause signals $(E\ e)\ \mathcal{P}$, we substitute all occurrences of e in $\mathcal{P}$ by exc and conjoin $\mathsf{typeof}(\mathsf{exc}) \preceq E \Rightarrow Tr(\mathcal{P})$ to Ψ_{exc}. signals_only is translated analogously.

The assignable-clause

The assignable $\bar{l}$ and modifies $\bar{l}$ clauses, where $\bar{l}$ refers to a set of locations, are synonymous. In JML, the understanding is that only the heap locations declared in the clause can be assigned to by the routine; that is, even if a method modifies a location and then re-establishes the old value, the location has to be mentioned in the assignable clause. We call this semantics 'assignable semantics'.

The assignable semantics can be translated by attaching an assert annotation $\mathsf{loc}(o, f) \in \bar{l}$ to every field update $o.f = v$ in the routine body if o was allocated already in the prestate of the routine. $\mathsf{loc}(o, f)$ yields the heap location denoted by object o and field identifier f. In addition, an assert annotation has to be attached to every routine call to check that its assignable clause is at least as restrictive as the one of the caller:

$$\forall o, f : \mathsf{loc}(o, f) \in \bar{l}_m \Rightarrow \mathsf{loc}(o, f) \in \bar{l}$$

where $\bar{l}_m$ is the set of locations that can be modified by the callee m. In this semantics, an assignable \nothing would lead to the special case which puts the assert annotation false in front of all field updates. Thus, the method cannot be verified if it contains a (reachable) field update. In this case, routine calls are annotated with the condition that the routine must not modify any heap locations. For assignable \everything, we can omit all assert annotations for field updates and routine calls as all of them are allowed.

Another way to understand it (especially the modifies keyword) is that the routine is not allowed to modify any heap location that is not mentioned in the clause. This would still allow temporary assignment of locations not mentioned in the clause, as long as the original value is assigned again. This is fine as long as we don't (indirectly) recursively call the routine itself and only have a single threaded program. We call this semantics 'modifiable semantics'.

To enforce the modifies semantics, we state in the postconditions of a routine that every heap location that is not in the modifiable set and that was already allocated in the prestate of the method still contains the old value. We conjoin

$$\forall o, f : \mathsf{loc}(o, f) \in \bar{l} \lor \neg\mathsf{Dom}(\mathsf{_pre_heap}, o) \lor \mathsf{Get}(\mathsf{heap}, o, f) = \mathsf{Get}(\mathsf{_pre_heap}, o, f)$$

to Ψ and Ψ_{exc}. If we encounter a modifies \nothing, $\bar{l}$ is empty which leads to the simpler version:

$$\forall o, f : \neg\mathsf{Dom}(\mathsf{_pre_heap}, o) \lor \mathsf{Get}(\mathsf{heap}, o, f) = \mathsf{Get}(\mathsf{_pre_heap}, o, f)$$

If we encounter a modifies \everything, we can simply ignore it for the translation as it does not restrict the routine anyhow in using heap locations.

In the current state of the work, JML data groups are not taken into account which

For simplicity, we chose to use the modifiable semantics here, even if this is not the JML semantics as it leads to less proof obligations. For the final version, we want to switch to the assignable semantics as used in JML. Another simplification is that we currently do not support data groups which means that the assignable clause only talks about a list of heap locations, the special case \everything which stands for all allocated heap locations, or \nothing which stands for no heap locations.

3.2.3 JML expressions

In general, the translation of JML expressions is straightforward. The operators are directly translated to the corresponding operators in FOL. The same applies to quantifiers and certain predicates like \type and \typeof. Still, there are some more interesting expressions that we discuss here.

The result expression

For each routine, a special result variable res of the return type of the routine is created. It is used each time the translation encounters a \result expression.

The old expression

In order to translate the JML \old ($\mathcal{P}$) expression, we need to have access to parts of the prestate of the current routine execution. For heap locations, this is done by saving a copy of the variable heap into _pre_heap. For each parameter p, we generate a ghost variable _pre_p that contains the value of the parameters in the prestate (The corresponding set annotations are attached to the first Java statement.

Having saved the interesting parts of the prestate, we can use it to translate the \old expression:

- field accesses $o.f$ inside \old expressions use _pre_heap instead heap in the Get function.
- parameter accesses p inside \old expressions lead to the use of the saved ghost variable _pre_p instead of the parameter p itself.

The fresh expression

For all heap locations $\bar{l}$ in \fresh ($\bar{l}$), we have to ensure that the location has not been allocated in the prestate. $Tr(\backslash\mathsf{fresh}(\bar{l}))$ is defines as follows:

$$\forall o, f : \mathsf{loc}(o, f) \in \bar{l} \Rightarrow \neg\mathsf{Dom}(\mathsf{_pre_heap}, \mathsf{loc}(o, f)) \wedge \mathsf{Dom}(\mathsf{heap}, \mathsf{loc}(o, f))$$

3.2.4 JML statements

In Section 3.1.2, we introduced the different kinds of local annotations that can be attached to a Java statement. For each JML statement, we have introduced an annotation type. assert $\mathcal{P}$ translates to the annotation assert $Tr(\mathcal{P})$, assume $\mathcal{P}$ translates to the annotation assume $Tr(\mathcal{P})$, and set $\mathcal{V} = e$ translates to the annotation set $Tr(\mathcal{V} = e)$.

Chapter 4

Preservation of proof obligations using a VCGen

In this chapter, we shall focus on the relation of the proof obligations over source programs and their respective non optimizing bytecode compilation. Particularly, we show that those are syntactically equivalent. In a PCC scenario such a relation enables the code producer to generate the certificate over the source code. As we pointed out earlier in the introduction, this is important when the certificate must be produced *interactively* which can be potentially the case when the security policy is non decidable. In such cases, reasoning over structured code is easier than reasoning over unstructured stack based code. We shall gradually investigate the relation of the proof obligations for three fragments of the language. In Section 4.1, we shall focus on a simple while language, Section 4.2 is dedicated to object oriented features and Section 4.3 takes into account an objected oriented language with exceptions.

4.1 Preservation of proof obligations for simple imperative programs

4.1.1 Verification condition generator for $\mathrm{JAVA}_{\mathcal{B}}$

The verification condition generator (VCGen) computes a set of verification conditions (logical propositions) such that their validity ensures the validity of a postcondition at the end of every run of a program. A postcondition is a proposition relating the final result to the initial values of the local variables. To define the VCGen, we need some logical annotations that cannot be inferred automatically, namely loop invariants. So we modify the syntax of a *while* instruction as follows: $\mathsf{while}_I(t)\{i\}$. Where I is a proposition, the loop invariant.

Definition 4.1.1 (Propositions) *The set of propositions is defined as follows:*

$$
\begin{array}{llcl}
\textit{logical expressions} & \bar{e} & ::= & \mathsf{res} \mid \bar{x} \mid x \mid c \mid \bar{e}\ op\ \bar{e} \\
\textit{logical tests} & \bar{t} & ::= & \bar{e}\ cmp\ \bar{e} \\
\textit{Propositions} & P & ::= & \bar{t} \mid \neg P \mid P \wedge P \mid P \vee P \mid P \Rightarrow P
\end{array}
$$

where $\bar{x}$ is a special variable representing the initial values of the variable x, and res *is a special value representing the final value of the evaluation of the program.*

An annotated program is a triple $(\mathcal{P}, \Phi, \Psi)$, where all while instructions in $\mathcal{P}$ are annotated with a loop invariant. In the proof-transforming compiler, these annotations are produced by the translation from JML to FOL as discussed in the previous chapter. Loop invariants should be valid at the entry and at the end of their corresponding loop. A loop invariant is a proposition that can only refer to the initial and current values of the local variables (not to the final result). Φ is the precondition of the program, which may only contain reference to the initial values of local variables. Ψ is the postcondition, which may only refer to the initial values of local variables and to the final result denoted by the special variable res.

$$\frac{}{\mathsf{wp}_{\mathcal{S}}(\mathsf{skip}, \psi) = \psi, \emptyset} \qquad \frac{}{\mathsf{wp}_{\mathcal{S}}(x := e, \psi) = \psi\{x \mapsto e\}, \emptyset}$$

$$\frac{\mathsf{wp}_{\mathcal{S}}(stmt_2, \psi) = \phi_2, \theta_2 \quad \mathsf{wp}_{\mathcal{S}}(stmt_1, \phi_2) = \phi_1, \theta_1}{\mathsf{wp}_{\mathcal{S}}(stmt_1; stmt_2, \psi) = \phi_1, \theta_1 \cup \theta_2}$$

$$\frac{\mathsf{wp}_{\mathcal{S}}(stmt_t, \psi) = \phi_t, \theta_t \quad \mathsf{wp}_{\mathcal{S}}(stmt_f, \psi) = \phi_f, \theta_f}{\mathsf{wp}_{\mathcal{S}}(\mathsf{if}(t)\{stmt_t\}\{stmt_f\}, \psi) = (t \Rightarrow \phi_t) \wedge (\neg t \Rightarrow \phi_t), \theta_t \cup \theta_f}$$

$$\frac{\mathsf{wp}_{\mathcal{S}}(stmt, I) = \phi, \theta}{\mathsf{wp}_{\mathcal{S}}(\mathsf{while}_I(t)\{stmt\}, \psi) = I, \{I \Rightarrow (t \Rightarrow \phi) \wedge (\neg t \Rightarrow \psi)\} \cup \theta}$$

$$\frac{\mathcal{P} = stmt; \mathsf{return}\ e \quad \mathsf{wp}_{\mathcal{S}}(stmt, \Psi\{\mathsf{res} \mapsto e\}) = \phi, \theta}{\mathsf{VCgen}_{\mathcal{S}}(\mathcal{P}, \Phi, \Psi) = \{\Phi \Rightarrow \phi\{\vec{x} \mapsto \vec{\bar{x}}\}\} \cup \theta}$$

Figure 4.1: Weakest precondition of the basic language

Figure 4.1 describes the verification condition generator. First, we define the weakest precondition of an instruction $\mathsf{wp}_{\mathcal{S}}(i, \psi)$ to ensure the postcondition ψ. The result is the precondition of the instruction, which should be valid before its execution, and a set of side conditions, which should be valid for any values of the local variables.

The set of verification conditions of an annotated program $\mathsf{VCgen}_{\mathcal{S}}(\mathcal{P}, \Phi, \Psi)$, where $\mathcal{P} = i; \mathsf{return}\ e$, is the side condition of the instruction i and the fact that the precondition of the program implies the weakest precondition of i. An annotated program $(\mathcal{P}, \Phi, \Psi)$ is correctly annotated if all generated conditions are provable.

Soundness

We denote by $\vdash \psi$ the fact that a proposition ψ is provable. This notation is naturally extended to sets of propositions.

Definition 4.1.2

- *An initial local memory $\bar{\rho}$ satisfies a precondition Φ if Φ with the variables $\bar{x}$ replaced by the corresponding values in $\bar{\rho}$ is a valid proposition:*

$$\bar{\rho} \models \Phi \stackrel{\text{def}}{\equiv} \vdash \Phi\{\vec{\bar{x}} \mapsto \overrightarrow{\bar{\rho}(x)}\}$$

- *An initial local memory $\bar{\rho}$ and a current local memory ρ satisfy a proposition ψ if ψ with the variables $\bar{x}$ replaced by the corresponding values in $\bar{\rho}$ and variables x replaced by the corresponding values in ρ is a valid proposition:*

$$\bar{\rho}, \rho \models \psi \stackrel{\text{def}}{\equiv} \vdash \psi\{\vec{\bar{x}} \mapsto \overrightarrow{\bar{\rho}(x)}\}\{\vec{x} \mapsto \overrightarrow{\rho(x)}\}$$

- *An initial memory $\bar{\rho}$ and a final result v satisfy a postcondition Ψ if Ψ with the variables $\bar{x}$ replaced by the corresponding values in $\bar{\rho}$ and the variable* res *replaced by v is a valid proposition:*

$$\bar{\rho}, v \models \Psi \stackrel{\text{def}}{\equiv} \vdash \Psi\{\vec{\bar{x}} \mapsto \overrightarrow{\bar{\rho}(x)}\}\{\mathsf{res} \mapsto v\}$$

Given an annotated program $(\mathcal{P}, \Phi, \Psi)$, which is correctly annotated $\vdash \mathsf{VCgen}_{\mathcal{S}}(\mathcal{P}, \Phi, \Psi)$, the soundness of the VCGen is expressed by the following property:

$$\left.\begin{array}{r} \bar{\rho} \models \Phi \\ \mathcal{P} : \bar{\rho} \Downarrow_{\mathcal{S}} v \end{array}\right\} \Rightarrow \bar{\rho}, v \models \Psi$$

In other terms, for every initial local memory $\bar{\rho}$ satisfying the precondition of the program, if the program evaluates to a final value v then $\bar{\rho}$ and v satisfy the postcondition of the program.

We do not prove this lemma, since the soundness of the VCGen for source programs will be a direct consequence of the soundness of the VCGen for bytecode and the preservation of proof obligations and the preservation of the semantics, see below.

4.1.2 The verification condition generator for $\mathrm{JVM}_{\mathcal{B}}$

The first difference compared to the VCGen for the source language $\mathrm{JAVA}_{\mathcal{B}}$ is the propositions. Propositions for $\mathrm{JVM}_{\mathcal{B}}$ must refer to elements in the operand stack. To allow this, we extend the expressions of the logic with a special variable **os** representing the current operand stack. $\bar{e} :: \bar{os}$ represents the stack $\bar{os}$ with the expression $\bar{e}$ on top, $\uparrow^k \bar{os}$ represents the stack $\bar{os}$ where the k top elements have been popped, and $\bar{os}[k]$ refers to the k+1-th element of the operand stack $\bar{os}$ (starting from the top). So, $\bar{os}[0]$ is the top element of the stack, $\bar{os}[1]$ is the second, etc.

Definition 4.1.3 (Bytecode Propositions) *The syntax of the bytecode propositions is defined by:*

$$\begin{array}{llcl} \textit{stack expressions} & \bar{os} & ::= & \mathbf{os} \mid \bar{e} :: \bar{os} \mid \uparrow^k \bar{os} \\ \textit{logical } \mathsf{bytecode} \textit{ expressions} & \bar{e} & ::= & \mathsf{res} \mid \bar{x} \mid x \mid c \mid \bar{e}\ op\ \bar{e} \mid \bar{os}[k] \\ \textit{logical tests} & \bar{t} & ::= & \bar{e}\ cmp\ \bar{e} \\ \mathsf{bytecode}\ \textit{Propositions} & \dot{P} & ::= & \bar{t} \mid \neg\dot{P} \mid \dot{P} \wedge \dot{P} \mid \dot{P} \vee \dot{P} \mid \dot{P} \Rightarrow \dot{P} \end{array}$$

where **os** *is a special variable representing the current operand stack.*

Intuitively, bytecode propositions should be understood as functions from local variables and an operand stack to logical propositions.

The second difference is the way of storing loop invariants (or annotations). In the source language $\mathrm{JAVA}_{\mathcal{B}}$, loop invariants are attached to *while* instruction. In the bytecode, loop invariants are stored in an external table $\Lambda : \mathcal{P}_c \to \dot{P} + \bot$, associating to some program point an annotation. Intuitively each time we reach an annotated program point, the annotation should be satisfied. An annotated bytecode program is a tuple $(\dot{\mathcal{P}}, \Phi, \Lambda, \Psi)$, where ϕ is the precondition of the program and ψ its postcondition.

At the level of the bytecode language, the predicate transformer is a partial function that computes, from a sufficiently annotated program, a fully annotated program in which all labels of the program have an explicit precondition attached to them. In order to ensure the decidability of VCGen computation, its domain is restricted to *well-annotated* programs. This domain can be characterized by an inductive and decidable definition and does not impose any specific structure on programs. Note that the verification condition generator does not ensure that well typedness of the bytecode. We actually assume that the code is "well behaved" in the sense that every time an instruction is executed the operand stack contains the right type and number of arguments.

Definition 4.1.4 (Well-annotated Program)

- *For all bytecode programs $\dot{\mathcal{P}}$ and program points k, the successor function $\mathsf{succ}_{\dot{\mathcal{P}}}(k)$ returns the set of program points that can be reached from one execution step starting from k:*

$$\mathsf{succ}_{\dot{\mathcal{P}}}(k) = \begin{cases} \emptyset & \textit{if}\ \dot{\mathcal{P}}[i] = \mathsf{return} \\ \{l\} & \textit{if}\ \dot{\mathcal{P}}[i] = \mathsf{goto}\ l \\ \{k+1, l\} & \textit{if}\ \dot{\mathcal{P}}[i] = \mathsf{if}\ cmp\ l \\ \{k+1\} & \textit{otherwise} \end{cases}$$

- *A program $\dot{\mathcal{P}}$ is closed if for any program point k in the domain $\dot{\mathcal{P}}$, all the successors of k are in the domain of $\dot{\mathcal{P}}$:*

$$\forall ks, k \in \mathsf{Dom}(\dot{\mathcal{P}}) \Rightarrow s \in \mathsf{succ}_{\dot{\mathcal{P}}}(k) \Rightarrow s \in \mathsf{Dom}(\dot{\mathcal{P}})$$

- *A program point k' is reachable from a label k in $\dot{\mathcal{P}}$ if $k = k'$ or if k' is the successor of a program point reachable from k:*

$$\frac{}{k \in \mathsf{reachable}_{\dot{\mathcal{P}},k}} \qquad \frac{k' \in \mathsf{reachable}_{\dot{\mathcal{P}},k} \quad k'' \in \mathsf{succ}_{\dot{\mathcal{P}}}(k')}{k'' \in \mathsf{reachable}_{\dot{\mathcal{P}},k}}$$

- *Given a* bytecode *program $\dot{\mathcal{P}}$ and an annotation table Λ, a program point k reaches annotated program points if the instruction at position k is annotated in Λ or if the instruction is a* return *(in that case the annotation is the postcondition), or if all its immediate successors reach annotated program points. More precisely,* $\mathsf{reachAnnot}_{\dot{\mathcal{P}},\Lambda}$ *is defined as the smallest set that satisfies the following conditions:*

$$\frac{\Lambda(k) = \dot{P}}{k \in \mathsf{reachAnnot}_{\dot{\mathcal{P}},\Lambda}} \qquad \frac{\dot{\mathcal{P}}[k] = \mathsf{return}}{k \in \mathsf{reachAnnot}_{\dot{\mathcal{P}},\Lambda}} \qquad \frac{\forall k' \in \mathsf{succ}_{\dot{\mathcal{P}}}(k), k' \in \mathsf{reachAnnot}_{\dot{\mathcal{P}},\Lambda}}{k \in \mathsf{reachAnnot}_{\dot{\mathcal{P}},\Lambda}}$$

- *An annotated program $(\dot{\mathcal{P}}, \phi, \Lambda, \psi)$ is well-annotated if it is closed and every reachable point from the starting point (i.e., label 0) reaches annotated labels.*

Given a well-annotated program, the verification condition generator is defined with two mutually recursive functions $\mathsf{wp}_{\mathcal{L}}(i)$ and $\mathsf{wp}_i(\dot{\mathcal{P}}[i])$. The function $\mathsf{wp}_{\mathcal{L}}(i)$ computes the weakest precondition of the program point i using the annotation table. If i is annotated ($\Lambda(i) = \dot{P}$), the weakest precondition is the annotation $\dot{P}$,; otherwise, the weakest precondition is the weakest precondition of the instruction at i, which is computed using the function wp_i. The function wp_i first computes the weakest precondition of all the successors of the instruction at i and then transforms the resulting condition depending on the instruction. Figure 4.2 defines these two functions.

Definition 4.1.5 *The set of verification condition of a well-annotated* bytecode *program* $\mathsf{VCgen}_{\mathcal{B}}(\dot{\mathcal{P}}, \Phi, \Lambda, \Psi)$ *is the the smallest set of propositions that contains the following implications:*

- *The precondition implies the weakest precondition of the starting point:*

$$(\Phi \Rightarrow \mathsf{wp}_{\mathcal{L}}(0)\{\vec{x} \mapsto \vec{\bar{x}}\})$$

- *For all annotated program point ($\Lambda(k) = \dot{P}$), the annotation $\dot{P}$ implies the weakest precondition of the instruction at k:*

$$\forall k, \Lambda(k) = \dot{P} \Rightarrow (\dot{P} \Rightarrow \mathsf{wp}_i(k))$$

Soundness

The key point to prove the soundness lemma is that if all conditions generated by the VCGen are valid then the weakest precondition of a program point $\mathsf{wp}_{\mathcal{L}}(i)$ implies the weakest precondition of its instruction $\mathsf{wp}_i(i)$.

Definition 4.1.6 (Interpretation of bytecode proposition) bytecode *propositions can be interpreted as predicates on* bytecode *states*

$$
\begin{array}{rcll}
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)\{\mathbf{os} \mapsto c :: \mathbf{os}\} & \text{if } \dot{\mathcal{P}}[k] = \mathsf{push}\ c \\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)\{\mathbf{os} \mapsto (\mathbf{os}[0]\ op\ \mathbf{os}[1]) ::\uparrow^2 \mathbf{os}\} & \text{if } \dot{\mathcal{P}}[k] = \mathsf{binop}\ op \\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)\{\mathbf{os} \mapsto x :: \mathbf{os}\} & \text{if } \dot{\mathcal{P}}[k] = \mathsf{load}\ x \\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(k+1)\{\mathbf{os}, x \mapsto \uparrow \mathbf{os}, \mathbf{os}[0]\} & \text{if } \dot{\mathcal{P}}[k] = \mathsf{store}\ x \\
\mathsf{wp}_i(k) & = & \mathsf{wp}_{\mathcal{L}}(l) & \text{if } \dot{\mathcal{P}}[k] = \mathsf{goto}\ l \\
\mathsf{wp}_i(k) & = & \begin{array}{l} \quad (\mathbf{os}[0]\ cmp\ \mathbf{os}[1] \Rightarrow \mathsf{wp}_{\mathcal{L}}(k+1)\{\mathbf{os} \mapsto \uparrow^2 \mathbf{os}\}) \\ \wedge \quad (\neg(\mathbf{os}[0]\ cmp\ \mathbf{os}[1]) \Rightarrow \mathsf{wp}_{\mathcal{L}}(l)\{\mathbf{os} \mapsto \uparrow^2 \mathbf{os}\}) \end{array} & \text{if } \dot{\mathcal{P}}[k] = \mathsf{if}\ cmp\ l \\
\mathsf{wp}_i(k) & = & \Psi\{\mathsf{res} \mapsto \mathbf{os}[0]\} & \text{if } \dot{\mathcal{P}}[k] = \mathsf{return} \\
 & = & & \\
\mathsf{wp}_{\mathcal{L}}(k) & = & \dot{P} & \text{if } \Lambda(k) = \dot{P} \\
\mathsf{wp}_{\mathcal{L}}(k) & = & \mathsf{wp}_i(k) & \text{if } \Lambda(k) = \bot
\end{array}
$$

Figure 4.2: WEAKEST PRECONDITION OF THE BASIC BYTECODE

- *The evaluation of a logical stack expression $\bar{os}$ and of a logical* bytecode *expression in an initial memory $\bar{\rho}$ and a current operand stack os are mutually defined by:*

$$
\frac{}{\bar{\rho}, os, \rho \vdash \mathbf{os} \mapsto os} \qquad \frac{\bar{\rho}, os, \rho \vdash \bar{e} \mapsto v \quad \bar{\rho}, os, \rho \vdash \bar{os} \mapsto os'}{\bar{\rho}, os, \rho \vdash \bar{e} :: \bar{os} \mapsto v :: os'} \qquad \frac{\bar{\rho}, os, \rho \vdash \bar{os} \mapsto v_0 :: \ldots :: v_{k-1} :: os'}{\bar{\rho}, os, \rho \vdash \uparrow^k \bar{os} \mapsto os'}
$$

$$
\frac{\bar{\rho}, os, \rho \vdash \bar{os} \mapsto os'}{\bar{\rho}, os, \rho \vdash \bar{os}[k] \mapsto os'[k]} \qquad \frac{}{\bar{\rho}, os, \rho \vdash \bar{x} \mapsto \bar{\rho}(x)} \qquad \frac{}{\bar{\rho}, os, \rho \vdash x \mapsto \rho(x)}
$$

$$
\frac{}{\bar{\rho}, os, \rho \vdash c \mapsto c} \qquad \frac{\bar{\rho}, os, \rho \vdash \bar{e}_1 \mapsto v_1 \quad \bar{\rho}, os, \rho \vdash \bar{e}_2 \mapsto v_2}{\bar{\rho}, os, \rho \vdash \bar{e}_1\ op\ \bar{e}_2 \mapsto v_1\ op\ v_2}
$$

- *This evaluation is naturally extended to* bytecode *propositions $\bar{\rho}, os, \rho \vdash P \mapsto P_v$, where P_v is a boolean formula, with the following rule for tests:*

$$
\frac{\bar{\rho}, os, \rho \vdash \bar{e}_1 \mapsto v_1 \quad \bar{\rho}, os, \rho \vdash \bar{e}_2 \mapsto v_2}{\bar{\rho}, os, \rho \vdash \bar{e}_1\ cmp\ \bar{e}_2 \mapsto v_1\ cmp\ v_2}
$$

- *An initial memory $\bar{\rho}$, a current operand stack os, and a current memory ρ validate a logical* bytecode *proposition P ($\bar{\rho}, os, \rho \models P$) if $\bar{\rho}, os, \rho \vdash P \mapsto P_v$ and P_v is provable.*

Lemma 6 *For all* bytecode *programs $\dot{\mathcal{P}}$, preconditions Φ, postconditions Ψ, and annotation tables Λ, if the proof obligations of $\dot{\mathcal{P}}$ are valid (i.e., $\vdash \mathsf{VCgen}_{\mathcal{B}}(\dot{\mathcal{P}}, \Phi, \Lambda, \Psi)$) then the following property holds:*

$$\bar{\rho}, \rho, os \models \mathsf{wp}_{\mathcal{L}}(k) \Rightarrow \bar{\rho}, \rho, os \models \mathsf{wp}_i(k)$$

Lemma 7 (Soundness for one execution step) *For all* bytecode *programs $\dot{\mathcal{P}}$, preconditions Φ, postconditions Ψ, and annotation tables Λ, if the proof obligations of $\dot{\mathcal{P}}$ are valid (i.e., $\vdash \mathsf{VCgen}_{\mathcal{B}}(\dot{\mathcal{P}}, \Phi, \Lambda, \Psi)$) then the following property holds:*

$$\left.\begin{array}{l} \bar{\rho}, \rho, os \models \mathsf{wp}_i(k) \\ \langle k,\ \rho,\ os\rangle \rightsquigarrow \langle k',\ \rho',\ os'\rangle \end{array}\right\} \Rightarrow \bar{\rho}, \rho', os' \models \mathsf{wp}_{\mathcal{L}}(k')$$

Lemma 8 (Soundness of the bytecode VCGen) *For all* bytecode *programs $\dot{\mathcal{P}}$, preconditions Φ, postconditions Ψ, and annotation tables Λ, if the proof obligations of $\dot{\mathcal{P}}$ are valid (i.e., $\vdash \mathsf{VCgen}_{\mathcal{B}}(\dot{\mathcal{P}}, \Phi, \Lambda, \Psi)$) then the following property holds:*

$$\left.\begin{array}{l} \bar{\rho}, \rho, os \models \mathsf{wp}_i(\dot{\mathcal{P}}[i]) \\ \langle k,\ \rho,\ os\rangle \Downarrow v \end{array}\right\} \Rightarrow \bar{\rho}, v \models \Psi$$

4.1.3 Relation between the verification calculi for $\text{JAVA}_\mathcal{B}$ and $\text{JVM}_\mathcal{B}$

Our goal is to show the preservation of proof obligations (PPO), i.e., given an annotated source program and its non optimized compilation, there exists an annotation table such that the verification conditions on bytecode and source code level will be syntactically equivalent:

$$\mathsf{VCgen}_\mathcal{S}(\mathcal{P}, \Phi, \Psi) = \mathsf{VCgen}_\mathcal{B}([\![\mathcal{P}]\!], \Phi, \Lambda, \Psi)$$

It is worth to note that the fact that the compiler is non optimizing is important for this result as in the presence of optimizations such an equivalence will not hold.

For the purposes of this section, we extend the compiler to annotated source programs. Only the compilation rule for the while statement changes: each time the compiler translates an annotated loop starting from position k ($k : [\![\mathsf{while}_I(e)\{c\}]\!]$), it inserts in the annotation table the invariant I at position k. The translation of the pre- and postcondition is the identity.

Lemma 9 (Well-annotated programs) *For all annotated source programs $(\mathcal{P}, \Phi, \Psi)$ that terminate by a return (i.e., $\mathcal{P} = i;$ $\mathsf{return}\, e$), the compiled version $(\dot{\mathcal{P}}, \Phi, \Lambda, \Psi)$ is well-annotated.*

This lemma implies that the VCGen on bytecode is defined. To prove preservation of proof obligations, we need two auxiliary lemmas. The first one expresses PPO for expressions, the second for instructions:

Lemma 10 (Preservation of proof obligations for expressions)
Given a well-annotated program $(i'; \mathsf{return}\ e', \Phi, \Psi)$ and its compiled version $(\dot{\mathcal{P}}, \Phi, \Lambda, \Psi)$. For all subexpressions e appearing in the program, if the sequence of code corresponding to the compilation of e starts at position k and terminates at position l (i.e., $l = k + |[\![e]\!]|$) and $\mathsf{wp}_\mathcal{L}(l) = \psi$ then $\mathsf{wp}_\mathcal{L}(k) = \psi\{\mathbf{os} \mapsto e :: \mathbf{os}\}$.

The proof is by induction on e. The two base cases are trivial, the case of binary operators needs some explanation. Let $e = e_1\ op\ e_2$. Its compiled sequence is

$$[\![e_2]\!];\ [\![e_1]\!];\ \mathsf{binop}\ op$$

Let $l_2 = k + |[\![e_2]\!]|$ and $l_1 = l_2 + |[\![e_1]\!]|$, so $l = l_1 + 1$. Since $\mathsf{wp}_\mathcal{L}(l) = \psi$, we have

$$\mathsf{wp}_\mathcal{L}(l_1) = \psi\{\mathbf{os} \mapsto \mathbf{os}[0]\ op\ \mathbf{os}[1] ::\uparrow^2 \mathbf{os}\}$$

By induction hypothesis on e_1

$$\begin{array}{rcl} \mathsf{wp}_\mathcal{L}(l_2) & = & \mathsf{wp}_\mathcal{L}(l_1)\{\mathbf{os} \mapsto e_1 :: \mathbf{os}\} \\ & = & \psi\{\mathbf{os} \mapsto (e_1 :: \mathbf{os})[0]\ op\ (e_1 :: \mathbf{os})[1] ::\uparrow^2 (e_1 :: \mathbf{os})\} \\ & = & \psi\{\mathbf{os} \mapsto e_1\ op\ \mathbf{os}[0] ::\uparrow^1 \mathbf{os}\} \end{array}$$

By induction hypothesis on e_2 we get the expected result

$$\begin{array}{rcl} \mathsf{wp}_\mathcal{L}(k) & = & \mathsf{wp}_\mathcal{L}(l_2)\{\mathbf{os} \mapsto e_2 :: \mathbf{os}\} \\ & = & \psi\{\mathbf{os} \mapsto e_1\ op\ e_2 :: \mathbf{os}\} \end{array}$$

Lemma 11 (Preservation of proof obligations for instructions)
Given a well-annotated program $(i'; \mathsf{return}\ e', \Phi, \Psi)$ and its compiled version $(\dot{\mathcal{P}}, \Phi, \Lambda, \Psi)$. For all sub-instructions $i \subseteq i'$, which are compiled starting from position k (i.e., $\dot{\mathcal{P}}[k..k + |[\![i]\!]|] = k : [\![i]\!]$) and for all postconditions ψ, if $\mathsf{wp}_\mathcal{S}(i, \psi) = \phi, \theta$ and $\mathsf{wp}_\mathcal{L}(k + |[\![i]\!]|) = \psi$ then the following properties hold:

- $\mathsf{wp}_\mathcal{L}(k) = \phi$
- *For all $C \in \theta$ there exists $k' \in [k..k + |[\![i]\!]|]$ and loop invariant I such that $\Lambda(k') = I$ and*

$$C = (I \Rightarrow \mathsf{wp}_i(k'))$$

The proof of this lemma is by induction over the instruction i. The two interesting case are for assignment and the while instruction. The assignment case is a direct consequence of PPO for expressions: by definition $k : [\![x := e]\!] = [\![e]\!];\ \mathsf{store}\ x$, since $\mathsf{wp}_{\mathcal{L}}(k + |[\![i]\!]|) = \mathsf{wp}_{\mathcal{L}}(k + |[\![e]\!]| + 1) = \psi$ we have

$$\mathsf{wp}_{\mathcal{L}}(k + |[\![e]\!]|) = \psi\{\mathbf{os}, x \mapsto \uparrow \mathbf{os}, \mathbf{os}[0]\}$$

and by lemma 10 we get

$$\begin{array}{rcl} \mathsf{wp}_{\mathcal{L}}(k) & = & \mathsf{wp}_{\mathcal{L}} k + |[\![e]\!]|\{\mathbf{os} \mapsto e :: \mathbf{os}\} \\ & = & (\psi\{\mathbf{os}, x \mapsto \uparrow \mathbf{os}, \mathbf{os}[0]\})\{\mathbf{os} \mapsto e :: \mathbf{os}\} \\ & = & \psi\{\mathbf{os}, x \mapsto \mathbf{os}, e\} \\ & = & \psi\{x \mapsto e\} \end{array}$$

For the while case $i = \mathsf{while}_I(t)\{i_1\}$ and $t = e_1\ cmp\ e_2$, we have

$$\begin{array}{rcl} k : [\![i]\!] & = & [\![e_2]\!];\ [\![e_1]\!];\ \mathsf{if}\ cmp\ k_2; k_1 : [\![i_1]\!];\ \mathsf{goto}\ k \\ \mathsf{wp}_{\mathcal{S}}(i, \psi) & = & I, \{I \Rightarrow (t \Rightarrow \phi_1) \wedge (\neg t \Rightarrow \psi)\} \cup \theta_1 \end{array}$$

with $\mathsf{wp}_{\mathcal{S}}(i_1, I) = \phi_1, \theta_1$, $l = k + |[\![e_1]\!]| + |[\![e_2]\!]|$ $k_1 = l + 1$ and $k_2 = k_1 + |[\![i_1]\!]| + 1 = k + |[\![i]\!]|$ Since $\Lambda(k) = I$ (by definition of the compiler), we have $\mathsf{wp}_{\mathcal{L}}(k) = I$ and the first property trivially holds. For the second, if $C \in \theta = \{I \Rightarrow (t \Rightarrow \phi_1) \wedge (\neg t \Rightarrow \psi)\} \cup \theta_1$, either $C \in \theta_1$ and the property holds by induction hypothesis on i_1 or $C = I \Rightarrow (t \Rightarrow \phi_1) \wedge (\neg t \Rightarrow \psi)$. In that case, we take $k' = k$ and we should prove that

$$\mathsf{wp}_i(k') = (t \Rightarrow \phi_1) \wedge (\neg t \Rightarrow \psi)$$

The WP of the last goto instruction is $\mathsf{wp}_{\mathcal{L}} k_1 + |[\![i_1]\!] = \mathsf{wp}_{\mathcal{L}} k = I$, so by induction hypothesis $\mathsf{wp}_{\mathcal{L}} k_1 = \phi_1$ and by hypothesis $\mathsf{wp}_{\mathcal{L}}(k_2) = \mathsf{wp}_{\mathcal{L}}(k + |[\![i]\!]|) = \psi$. The WP of the branching instruction is

$$\begin{array}{rcl} \mathsf{wp}_{\mathcal{L}}(l) & = & (\mathbf{os}[0]\ cmp\ \mathbf{os}[1] \Rightarrow \mathsf{wp}_{\mathcal{L}}(k_1)\{\mathbf{os} \mapsto \uparrow^2 \mathbf{os}\}) \wedge (\neg(\mathbf{os}[0]\ cmp\ \mathbf{os}[1]) \Rightarrow \mathsf{wp}_{\mathcal{L}}(k_2)\{\mathbf{os} \mapsto \uparrow^2 \mathbf{os}\}) \\ & = & (\mathbf{os}[0]\ cmp\ \mathbf{os}[1] \Rightarrow \phi_1\{\mathbf{os} \mapsto \uparrow^2 \mathbf{os}\}) \wedge (\neg(\mathbf{os}[0]\ cmp\ \mathbf{os}[1]) \Rightarrow I\{\mathbf{os} \mapsto \uparrow^2 \mathbf{os}\}) \\ & = & (\mathbf{os}[0]\ cmp\ \mathbf{os}[1] \Rightarrow \phi_1 \wedge (\neg(\mathbf{os}[0]\ cmp\ \mathbf{os}[1]) \Rightarrow I) \end{array}$$

For the last equation, we use the fact that $\mathbf{os}$ does not appear in ϕ_1 and I (they are source propositions). Then, using lemma 10, we conclude:

$$\mathsf{wp}_{\mathcal{L}}(k) = (e_1\ cmp\ e_2 \Rightarrow \phi_1 \wedge (\neg(e_1\ cmp\ e_2) \Rightarrow I)$$

Lemma 12 (Preservation of proof obligations (PPO)) *The sets of verification conditions of a well-annotated source program $(\mathcal{P}, \Phi, \Psi)$ terminating by a return, and its compiled version $(\dot{\mathcal{P}}, \Phi, \Lambda, \Psi)$ are equal:*

$$\mathsf{VCgen}_{\mathcal{S}}(\mathcal{P}, \Phi, \Psi) = \mathsf{VCgen}_{\mathcal{B}}(\dot{\mathcal{P}}, \Phi, \Lambda, \Psi)$$

Using the fact that the compiler preserves the semantics of programs, the soundness of the verification condition generator for bytecode and PPO, we can derive soundness of the source verification condition generator.

Corolary 4.1.7 (Soundness of $\mathsf{VCgen}_{\mathcal{S}}$) *If $\vdash \mathsf{VCgen}_{\mathcal{S}}(\mathcal{P}, \Phi, \Psi)$ then for all initial memory ρ_0 satisfying Φ, if $\mathcal{P} : \rho_0 \Downarrow_{\mathcal{S}} \rho, v$ then $\rho_0, \rho \vdash \Psi$.*

4.2 Preservation of proof obligations for objects and methods

Now that we have introduced the basic concepts of the proof preserving compilation, we are ready to consider several more realistic extensions of the language. In this section particularly, we shall consider a language with object-oriented features.

4.2.1 Method specification and behavior subtyping

In $\text{JAVA}_\mathcal{O}$ and $\text{JVM}_\mathcal{O}$ a program is a set of classes, we have to change the notion of program specification.

In an object-oriented setting, the basic execution entities are methods. A method is provided with a specification (or also contract), i.e., two propositions describing under what conditions a method may be called (the method precondition) and what is guaranteed by the method (the method postcondition). Thus, we shall provide a program $\mathcal{P}_o$ with a table Γ describing the contract of every method in the program. When we want to express that the specification found in Γ for method m is the precondition Φ and Ψ, we shall write $\Gamma(\mathsf{m}, \Phi, \Psi)$.

The notion of program validity relies on the validity of every method in the program and is as follows:

Definition 4.2.1 (Validity of programs)
A program $\mathcal{P}_o$ is valid if every method declared in a class in $\mathcal{P}_o$ is valid w.r.t. its specification.

Method validity w.r.t. method specifications Assuming that the specification of the method m consists of the precondition Φ and the postcondition Ψ (i.e., $\Gamma(\mathsf{m}, \Phi, \Psi)$), we say that m is valid w.r.t. its specification if the execution of the method in a state satisfying the precondition leads to a value satisfying the postcondition. This is formally stated in the following definition:

Definition 4.2.2 (Validity of methods)

- *Method validity for* $\text{JAVA}_\mathcal{O}$

$$\begin{array}{l} \models \Gamma(\mathsf{m}, \Phi, \Psi) \stackrel{\text{def}}{\equiv} \\ \quad \forall h\ r\ v\ v'\ h'.\ [\mathsf{m}, r, v, h] \Downarrow_\mathcal{S} (v', h') \land (\{\mathsf{this} \mapsto r, arg \mapsto v\}, h) \models \Phi \Rightarrow \\ \quad\quad (\{\mathsf{this} \mapsto r, arg \mapsto v\}, h), (h', v') \models \Psi \end{array}$$

- *Method validity for* $\text{JVM}_\mathcal{O}$

$$\begin{array}{l} \models \Gamma(\mathsf{m}, \Phi, \Psi) \stackrel{\text{def}}{\equiv} \\ \quad \forall h\ r\ v\ v'\ h'.\ \langle 1, \{\mathsf{this} \mapsto r, arg \mapsto v\}, \mathsf{nil}, h\rangle \Downarrow_\mathsf{m} (v', h') \land (\{\mathsf{this} \mapsto r, arg \mapsto v\}, h) \models \Phi \Rightarrow \\ \quad\quad (\{\mathsf{this} \mapsto r, arg \mapsto v\}, h), (h', v') \models \Psi \end{array}$$

Method validity in the presence of virtual method calls In the presence of method overriding, it is not clear at compile time which of the methods will be executed, the overridden method or which of its overriding methods. This depends on the dynamic type of the receiver object which in general cannot be determined statically. This implies that an overriding method m_1 may be executed whenever their is a static call to method m_2 overriden by m_1. Intuitively, we would expect that m_1 may be executed under the same conditions as m_2, and that m_2 must provide the same guarantees as the overridden method m_1; this property is called behavioral subtyping [20]. This is formally stated in the following definition:

Definition 4.2.3 (Behavioral subtyping validity of methods)
Let m_2 be a method declared in class C_2 such that C_1 is an ancestor class of C_2 in the program $\mathcal{P}_o$. Moreover, let m_2 override method m_1. If the specifications of m_2 and m_1 are $\Gamma(\mathsf{m}_2, \Phi_2, \Psi_2)$ and $\Gamma(\mathsf{m}_1, \Phi_1, \Psi_1)$, respectively, then we say that method m_2 is a behavioral subtype of m_1 if the following two conditions hold:

1. $\forall \rho_0\ h_0,\ (\rho_0,\ h_0) \models \Phi_1 \Rightarrow (\rho_0,\ h_0) \models \Phi_2$
2. $\forall \rho_0\ h_0\ h\ v,\ (\rho_0,\ h_0) \models \Phi_1 \Rightarrow (\rho_0,\ h_0), (h, v) \models \Psi_2 \Rightarrow (\rho_0,\ h_0), (h, v) \models \Psi_1$

A first option to statically ensure this property is to look for all the possible methods that can be actually executed at a method call (find all the methods that are overriding the method) and check if their specifications respect the respective pre- and postcondition of the method call. But this is a non-modular solution, as the whole program should be re-verified every time the program is extended with a new subclass.

$$\frac{\Gamma(\mathsf{m}, \Phi, \Psi) \quad \mathsf{m} \in \mathsf{C} \quad \Gamma(\mathsf{m}', \Phi', \Psi') \quad \mathsf{m}' \in \mathsf{C} \quad \mathsf{Superclass}(\mathsf{C}, \mathsf{C}') \quad \mathsf{overrides}(\mathsf{m}', \mathsf{m})}{\Phi' \Rightarrow \Phi \in \mathsf{VCbs}_{\mathcal{O}}(\Gamma)}$$

$$\frac{\Gamma(\mathsf{m}, \Phi, \Psi) \quad \mathsf{m} \in \mathsf{C} \quad \Gamma(\mathsf{m}', \Phi', \Psi')\mathsf{m}' \in \mathsf{C}' \quad \mathsf{Superclass}(\mathsf{C}, \mathsf{C}') \quad \mathsf{overrides}(\mathsf{m}', \mathsf{m})}{\Psi \Rightarrow \Psi' \in \mathsf{VCbs}_{\mathcal{O}}(\Gamma)}$$

Figure 4.3: CONDITIONS FOR ENFORCING BEHAVIORAL SUBTYPING IN JAVA$_{\mathcal{O}}$

Another possibility is to use specification inheritance [13] which provides a modular and sound reasoning in the presence of virtual method calls. This solution is also adopted in JML [37]. The technique consists in synthesizing method specifications from its proper specification (e.g., provided by the developer) and the specifications of the methods it overrides. In particular, the synthesized method precondition is the disjunction of its specified precondition and the preconditions of its overridden methods. The postcondition of a method states that if its specified precondition holds in the initial state then its specified postcondition holds in the final state and similarly for the specified pre- and postcondition of every overridden method. Such synthesized specifications meet the condition of behavioral subtyping without the need of additional verification conditions but however may lead to prove large specifications against which a method should be verified if the chain of overriding methods is long.

Here, we shall adopt another approach of enforcing behavioral subtyping as shown in Fig. 4.3: for the specification table Γ of program $\mathcal{P}_o$, we require that for every overriding method in a class C there is a co- and contravariance relation between its pre- and postcondition and the pre- and postcondition of the overridden method in the direct super class C' of $\mathsf{C}$for which we use the notation $\mathsf{Superclass}(\mathsf{C}, \mathsf{C}')$. This is sufficient to establish that a method in the presence of virtual calls is a correct behavioral subtype of the methods it overrides by transitivity of the logical implication. This technical is used at source and bytecode level.

At source level and at bytecode level the verification conditions of a program are split in two parts:

- A first part to ensure behavior subtyping checker

$$\mathsf{VCbs}_{\mathcal{O}}(\Gamma)$$

  This part is common to the source and bytecode level

- A second to check that the implementation of each method satisfies its specification.

Since the part of the verification conditions ensuring the behavior subtyping is common to the two language level, our goal is now to show the preservation of proof obligations of the second part.

4.2.2 Verification condition generator for JAVA$_{\mathcal{O}}$

Program specification for JAVA$_{\mathcal{O}}$ programs

The specification language for JAVA$_{\mathcal{O}}$ is an extension of the one of JAVA$_{\mathcal{B}}$ but it take into account heap expressions.

Definition 4.2.4 (Propositions language of JAVA$_{\mathcal{O}}$)

The set of propositions is defined as follows:

$$\begin{array}{llcl}
\textit{logical expressions} & \bar{e} & ::= & \mathsf{res} \mid \bar{x} \mid x \mid v \mid c \mid \bar{e}\ op\ \bar{e} \mid H(\bar{e}.\mathsf{f}) \\
\textit{heap expressions} & H & ::= & H\{\bar{e}.\mathsf{f} \mapsto \bar{e}\} \mid \bar{\mathbf{h}} \mid \mathbf{h} \mid h \\
\textit{logical tests} & \bar{t} & ::= & \bar{e}\ cmp\ \bar{e} \\
\textit{Propositions} & P & ::= & \bar{t} \mid \neg P \mid P \wedge P \mid P \vee P \mid P \Rightarrow P \\
 & & \mid & \forall v.\ P \mid \forall h.\ P \mid \mathsf{New}(H, C) = (h, v)
\end{array}$$

In the new constructs for expressions, v are variables which are bind by a for all expression and $H(\bar{e}.\mathsf{f})$ stand for the value of the field f associated to the expression $\bar{e}$ in the heap H. Heap expressions are either a heap variable (bind by a for all), or the special variable $\bar{\mathbf{h}}$ representing the initial heap (at the beginning of the evaluation of the method or the special variable $\mathbf{h}$ representing the current heap or $H\{\bar{e}_1.\mathsf{f} \mapsto \bar{e}_2\}$ which stand for the heap H where the field f of the expression $\bar{e}_1$ has been updated with the value $\bar{e}_2$. Proposition are also extended with two *for all* expressions, one for quantifying over values and the other over heaps. Thus, we introduce a special predicate $\mathsf{New}(H, C) = (h, v)$ which means that the variable h represents the heap resulting of a new allocation of a object of class C in the heap H and v is a reference to the newly allocated object.

We also introduce special logical variables, which, as we shall see later, are abstractions for expression values. Those variables are taken from a set $\mathcal{LV}$ which is disjoint from the set of the program variables. To give an intuition about $\mathcal{LV}$, an assertion may mention a logical variable if it refers to intermediate states of expression evaluation. Thus, method pre- and postconditions must not contain such logical variables. Actually, as we shall see later, those variables will appear only in the intermediate calculations of the verification condition generator.

Validity of assertions is defined in a similar way as in the previous section, but it is now parametrised by four components, the initial local and global memory and the current local and global memories:

Definition 4.2.5 (Validity of pre- and postconditions)

- *An initial state* $(\rho_0,\ h_0)$ *satisfies a precondition* Φ *if the interpretation of* Φ *in this state holds:*

$$(\rho,\ h) \models \Phi \ \stackrel{\text{def}}{\equiv} \vdash \Phi\{\vec{\bar{x}}, \bar{\mathbf{h}} \mapsto \rho_0(\vec{x}), h_0\}$$

- *An initial state* $(\rho_0,\ h_0)$ *and a current state* $(\rho,\ h)$ *satisfy the proposition* ψ *if the interpretation of* ψ *in these states holds:*

$$(\rho_0,\ h_0), (\rho,\ h) \models\!\vdash \psi \ \stackrel{\text{def}}{\equiv}\ \psi\{\vec{\bar{x}}, \bar{\mathbf{h}}, \vec{x}, \mathbf{h} \mapsto \rho_0(\vec{x}), h_0, \rho(\vec{x}), h\}$$

- *An initial state* $(\rho_0,\ h_0)$*, a final memory* h *and a final value* v *satisfy the postcondition* ψ *if the interpretation of* ψ *in these states and for this value holds:*

$$(\rho_0,\ h_0), (h, v) \models \psi \ \stackrel{\text{def}}{\equiv} \vdash \psi\{\vec{\bar{x}}, \bar{\mathbf{h}}, \mathbf{h}, \mathsf{res} \mapsto \rho_0(\vec{x}), h_0, h, v\}$$

Verification condition generator for $\text{JAVA}_{\mathcal{O}}$

Fig. 4.4 shows the rules for generating verification conditions for $\text{JAVA}_{\mathcal{O}}$. Now that evaluation of expressions may change the program state (e.g., instance creation expression and method invocation), the verification condition generator becomes more complex.

First, we define the weakest precondition $\mathsf{wp}_{\mathcal{O}}(e, \psi)_v$ of an expression e to ensure a postcondition ψ the variable v as to be understand as a binder which links the value of the expression e in ψ. If the expression e is the variable x the weakest precondition is $\psi\{v \mapsto x\}$. If it is a binary operation then the WP is a composition of the WP of the two expressions, the order of the composition is important due to the possible side effect of expressions. For method invocation, the WP lookup in the method specification table the

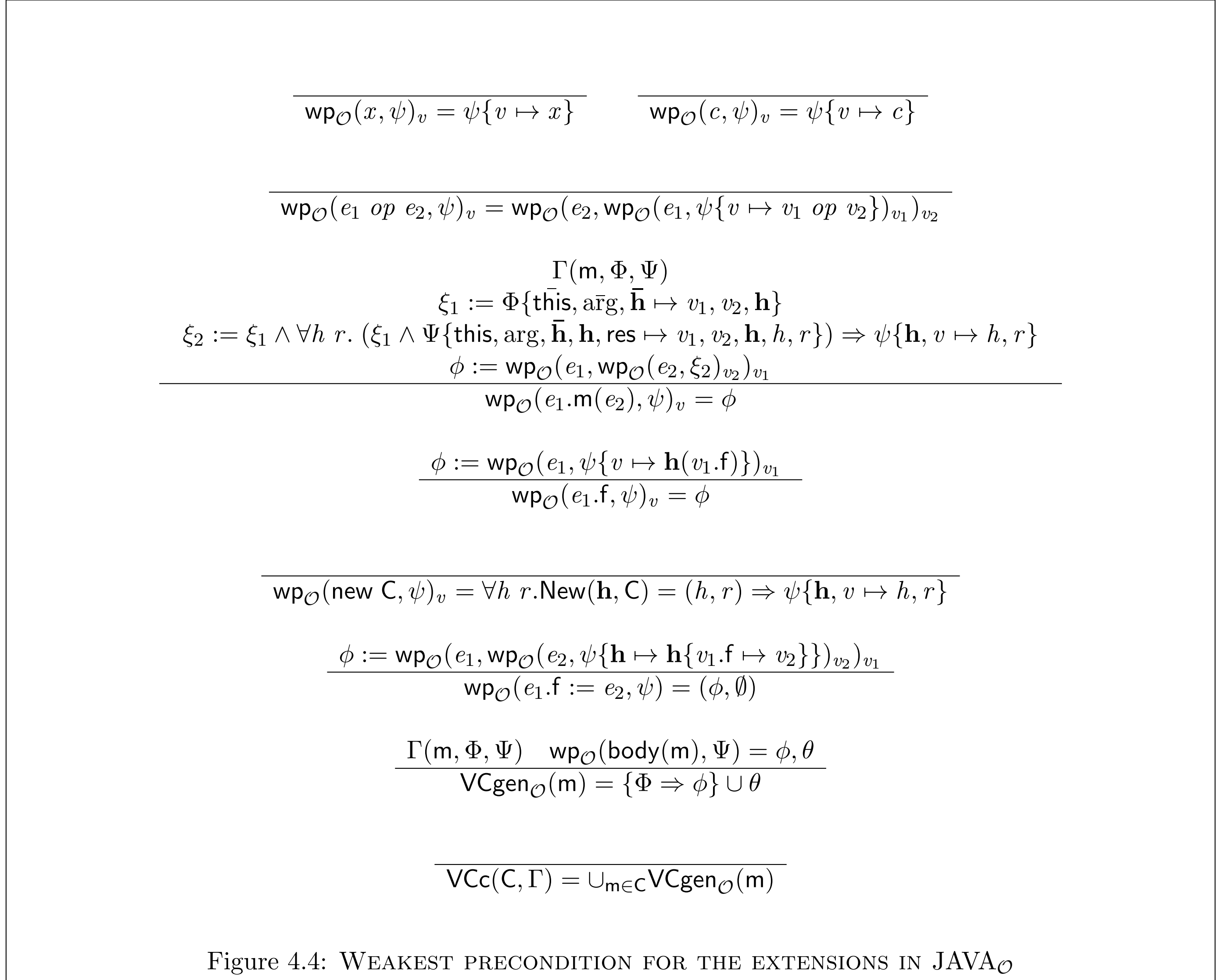


Figure 4.4: WEAKEST PRECONDITION FOR THE EXTENSIONS IN JAVA$_{\mathcal{O}}$

specification associated to the method m in the class C (the static type of the expression e_1), and use it to build the precondition of the call. The heap after the call and the result are universally quantified.

Second, we define the weakest precondition for instruction $\mathsf{wp}_{\mathcal{O}}(i, \psi)$, as for JAVA$_{\mathcal{B}}$ the result is a precondition for the instruction i and a set of side conditions. The rule for field assignment states that the postcondition ψ must hold in the post-state of the field assignment, i.e. after the evaluation of e_1 and e_2 for their values v_1 and v_2.

The verification conditions for a method implementation $\mathsf{VCgen}_{\mathcal{O}}(\mathsf{m})$ is computed using the specification given by Γ and weakest precondition of the method body. Those verification conditions for method implementations are then used to compute the set of verification condition of a class $\mathsf{VCc}(\mathsf{C}, \Gamma)$.

The verification conditions for a program $\mathcal{P}_o$ with a specification table Γ express that $\mathcal{P}_o$ is correct if every method in every class in $\mathcal{P}_o$ is correct w.r.t. Γ and verify the behavior subtyping condition.

Definition 4.2.6 (Verification conditions for JAVA$_{\mathcal{O}}$)

- *The verification conditions of the method implementations of a program are defined as:*

$$\mathsf{VCp}_{\mathcal{O}}(\mathcal{P}_o, \Gamma) = \cup_{\mathsf{C} \in \mathcal{P}_o} \mathsf{VCc}(\mathsf{C}, \Gamma)$$

- *The verification conditions of a program are:*

$$\mathsf{VCp}_{\mathcal{O}}(\mathcal{P}_o, \Gamma) \cup \mathsf{VCbs}_{\mathcal{O}}(\Gamma)$$

The soundness statement of the verification condition generator $\mathsf{VCgen}_{\mathcal{O}}$ is expressed as follows:

Theorem 4.2.7 (Soundness of the verification condition generator for JAVA$_{\mathcal{O}}$**)**
Let $\mathcal{P}_o$ be a JAVA$_{\mathcal{O}}$ *program provided with its specification table* Γ*. If the following conditions hold:*

1. *the verification conditions over methods are valid, i.e.,* $\vdash \mathsf{VCp}_{\mathcal{O}}(\mathcal{P}_o, \Gamma)$
2. *overriding methods are proper behavioral subtypes, i.e.,* $\vdash \mathsf{VCbs}_{\mathcal{O}}(\Gamma)$

then $\mathcal{P}_o$ is valid in the sense of Definition 4.2.1

4.2.3 Verification conditions generator for JVM$_{\mathcal{O}}$

Like in the source case, a JVM$_{\mathcal{O}}$ program $\mathcal{P}_o$ is specified with a global specification table Γ, where $\Gamma(\mathsf{m}, \Phi, \Psi)$ means that the precondition and postcondition for method m are Φ and Ψ, respectively. We also need a local specification table Λ containing the local invariants of each methods. This local table can be see as a map associating to each method a JVM$_{\mathcal{B}}$ annotation table, i.e. $\Lambda_{\mathsf{m}}(k) = I$ means that the program point at position k in the bytecode sequence corresponding to the method m is annotated with the invariant I, whereas $\Lambda_{\mathsf{m}}(k) = \bot$ means that the program point k in m is not annotated.

As for JAVA$_{\mathcal{B}}$ and JVM$_{\mathcal{B}}$, the specification language of JAVA$_{\mathcal{O}}$ is extended to allows stack expressions.

Definition 4.2.8 (Specification language of JVM$_{\mathcal{O}}$**)**
The set of propositions is defined by:

$$\begin{array}{llcl}
\textit{stack expressions} & \bar{os} & ::= & \mathbf{os} \mid \bar{e} :: \bar{os} \mid \uparrow^k \bar{os} \\
\textit{logical expressions} & \bar{e} & ::= & \mathsf{res} \mid \bar{x} \mid x \mid v \mid c \mid \bar{e}\ op\ \bar{e} \mid H(\bar{e}.f) \mid \bar{os}[k] \\
\textit{heap expressions} & H & ::= & H\{\bar{e}.f \mapsto \bar{e}\} \mid \bar{\mathbf{h}} \mid \mathbf{h} \mid h \\
\textit{logical tests} & \bar{t} & ::= & \bar{e}\ cmp\ \bar{e} \\
\textit{Propositions} & P & ::= & \bar{t} \mid \neg P \mid P \wedge P \mid P \vee P \mid P \Rightarrow P \\
 & & \mid & \forall v.\ P \mid \forall h.\ P \mid \mathsf{New}(H, C) = (h, v)
\end{array}$$

Verification condition generator for JVM$_{\mathcal{O}}$

The verification condition generator for the object-oriented instructions is given in Fig. 4.5. The generation of verification conditions is based on the predicate transformer function $\mathsf{wp}_i^{\mathsf{m}}$. Let us look at some of the rules in more detail. The rule for the method invocation generates a precondition for the instruction which expresses the following: in the prestate, the precondition of the invoked method must hold, where the value of the argument is initialized with the top stack element and the receiver object this is initialized with the second stack top element. The postcondition of the method must imply the precondition of the successor instruction (calculated by $\mathsf{wp}_l^{\mathsf{m}}$) for any return value and where the receiver object and the argument are initialized with the values from the operand stack.

As for JAVA$_{\mathcal{O}}$, the set of verification conditions of a bytecode program is the union of the verification condition due to the behavior subtyping and to the verification conditions of each classes of the program.

Definition 4.2.9 (Verification conditions of a bytecode program)

- *The verification conditions of the method implementations of a bytecode program are defined as:*

$$\mathsf{VCp}_{\mathcal{O}}^{bc}(\mathcal{P}_o, \Gamma) = \cup_{\mathsf{C} \in \mathcal{P}_o} \mathsf{VCc}_{\mathcal{O}}^{bc}(\mathsf{C}, \Gamma)$$

- *The verification conditions of a bytecode program are:*

$$\mathsf{VCp}_{\mathcal{O}}^{bc}(\mathcal{P}_o, \Gamma) \cup \mathsf{VCbs}_{\mathcal{O}}(\Gamma)$$

$$\frac{\begin{array}{c} P_{\mathsf{m}}[k] = \mathsf{invoke}\ \mathsf{m}' \qquad \Gamma(\mathsf{m}', \Phi, \Psi) \qquad \mathsf{wp}^{\mathsf{m}}_{l}(k+1) = \psi \\ \xi_1 = \Phi\{\bar{\mathsf{this}}, \bar{\mathrm{arg}}, \bar{\mathbf{h}} \mapsto \mathbf{os}[1], \mathbf{os}[0], \mathbf{h}\} \\ \phi = \xi_1 \wedge \forall h\ r.\ \xi_1 \wedge \Psi\{\bar{\mathsf{this}}, \bar{\mathrm{arg}}, \bar{\mathbf{h}}, \mathbf{h}, \mathsf{res} \mapsto \mathbf{os}[1], \mathbf{os}[0], \mathbf{h}, h, r\} \Rightarrow \psi\{\mathbf{h}, \mathbf{os} \mapsto h, r ::\uparrow^2 \mathbf{os}\} \end{array}}{\mathsf{wp}^{\mathsf{m}}_{i}(k) = \phi}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{putfield}\ \mathsf{f}}{\mathsf{wp}^{\mathsf{m}}_{i}(k) = \mathsf{wp}^{\mathsf{m}}_{l}(k+1)\{\mathbf{h}, \mathbf{os} \mapsto \mathbf{h}\{\mathbf{os}[1].\mathsf{f} \mapsto \mathbf{os}[0]\}, \uparrow^2 \mathbf{os}\}}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{getfield}\ \mathsf{f}}{\mathsf{wp}^{\mathsf{m}}_{i}(k) = \mathsf{wp}^{\mathsf{m}}_{l}(k+1)\{\mathbf{os} \mapsto \mathbf{h}(\mathbf{os}[0].\mathsf{f}) ::\uparrow \mathbf{os}\}}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{new}\ \mathsf{C} \qquad \mathsf{wp}^{\mathsf{m}}_{l}(k+1) = \phi}{\mathsf{wp}^{\mathsf{m}}_{i}(k) = \forall h\ r.\ \mathsf{New}(\mathbf{h}, \mathsf{C}) = (h, r) \Rightarrow \phi\{\mathbf{h}, \mathbf{os} \mapsto h, r :: \mathbf{os}\}}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{push}\ c}{\mathsf{wp}^{\mathsf{m}}_{i}(k) = \mathsf{wp}^{\mathsf{m}}_{l}(k+1)\{\mathbf{os} \mapsto c :: \mathbf{os}\}} \qquad \frac{P_{\mathsf{m}}[k] = \mathsf{binop}\ op}{\mathsf{wp}^{\mathsf{m}}_{i}(k) = \mathsf{wp}^{\mathsf{m}}_{l}(k+1)\{\mathbf{os} \mapsto (\mathbf{os}[0]\ op\ \mathbf{os}[1]) ::\uparrow^2 \mathbf{os}\}}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{load}\ x}{\mathsf{wp}^{\mathsf{m}}_{i}(k) = \mathsf{wp}^{\mathsf{m}}_{l}(k+1)\{\mathbf{os} \mapsto x :: \mathbf{os}\}} \qquad \frac{P_{\mathsf{m}}[k] = \mathsf{store}\ x}{\mathsf{wp}^{\mathsf{m}}_{i}(k) = \mathsf{wp}^{\mathsf{m}}_{l}(k+1)\{\mathbf{os}, x \mapsto \uparrow \mathbf{os}, \mathbf{os}[0]\}}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{goto}\ l}{\mathsf{wp}^{\mathsf{m}}_{i}(k) = \mathsf{wp}^{\mathsf{m}}_{l}(l)} \qquad \frac{P_{\mathsf{m}}[k] = \mathsf{if}\ cmp\ l}{\mathsf{wp}^{\mathsf{m}}_{i}(k) = \begin{array}{ll} & (\mathbf{os}[0]\ cmp\ \mathbf{os}[1] \Rightarrow \mathsf{wp}^{\mathsf{m}}_{l}(k+1)\{\mathbf{os} \mapsto \uparrow^2 \mathbf{os}\}) \\ \wedge & (\neg(\mathbf{os}[0]\ cmp\ \mathbf{os}[1]) \Rightarrow \mathsf{wp}^{\mathsf{m}}_{l}(l)\{\mathbf{os} \mapsto \uparrow^2 \mathbf{os}\}) \end{array}}$$

$$\frac{P_{\mathsf{m}}[k] = \mathsf{return} \quad \Gamma(\mathsf{m}, \Phi, \Psi)}{\mathsf{wp}^{\mathsf{m}}_{i}(k) = \Psi\{\mathsf{res} \mapsto \mathbf{os}[0]\}}$$

$$\frac{\Lambda_{\mathsf{m}}(k) = \dot{P}}{\mathsf{wp}^{\mathsf{m}}_{l}(k) = \dot{P}} \qquad \frac{\Lambda_{\mathsf{m}}(k) = \perp}{\mathsf{wp}^{\mathsf{m}}_{l}(k) = \mathsf{wp}^{\mathsf{m}}_{i}(k)}$$

$$\frac{\Gamma(\mathsf{m}, \Phi, \Psi)}{\mathsf{VCgen}^{bc}_{\mathcal{O}}(\mathsf{m}) = \{\psi \Rightarrow \mathsf{wp}^{\mathsf{m}}_{i}(k) \mid \Lambda_{\mathsf{m}}(k) = \psi\} \cup \{\Phi \Rightarrow \mathsf{wp}^{\mathsf{m}}_{i}(1)\}}$$

$$\frac{}{\mathsf{VCc}^{bc}_{\mathcal{O}}(\mathsf{C}, \Gamma) = \cup_{\mathsf{m} \in \mathsf{C}} \mathsf{VCgen}^{bc}_{\mathcal{O}}(\mathsf{m})}$$

Figure 4.5: VERIFICATION CONDITION GENERATOR FOR $\mathrm{JVM}_{\mathcal{O}}$

The soundness of the verification condition generator states that if the the verification conditions generated for a program are valid, then the program is valid in the sense of Definition 4.2.1:

Theorem 4.2.10 (Soundness of the verification condition generator for $\mathrm{JVM}_{\mathcal{O}}$)
Let $\mathcal{P}_o$ be a $\mathrm{JVM}_{\mathcal{O}}$ program provided with a specification table Γ. If the following conditions hold:

1. *the verification conditions over methods are valid, i.e.,* $\vdash \mathsf{VCp}^{bc}_{\mathcal{O}}(\mathcal{P}_o, \Gamma)$
2. *overriding methods are proper behavioral subtypes, i.e.,* $\vdash \mathsf{VCbs}_{\mathcal{O}}(\Gamma)$

then $\mathcal{P}_o$ is valid in the sense of Definition 4.2.1

The proof is similar to the case of $\mathrm{JVM}_{\mathcal{B}}$ the only new difficulty is the case of method invocation, where we use the fact that the condition ensuring behavior subtyping are valid, and the property of the lookup function which return a overriding method.

4.2.4 Relation between the verification calculi for $\mathrm{JAVA}_{\mathcal{O}}$ and $\mathrm{JVM}_{\mathcal{O}}$

Similarly to the simple source and bytecode languages, the verification conditions for a $\mathrm{JAVA}_{\mathcal{O}}$ program against a specification table Γ and its compilation in $\mathrm{JVM}_{\mathcal{O}}$ against the same specification table extended with the local annotation table containing the loop invariants, are syntactically equivalent under the assumption that the local annotations (i.e. loop invariants) are compiled in a proper way. The formal statement follows.

Theorem 4.2.11 (Equivalence between $\mathsf{VCp}_{\mathcal{O}}$ and $\mathsf{VCp}^{bc}_{\mathcal{O}}$)
For all $\mathrm{JAVA}_{\mathcal{O}}$ program $\mathcal{P}_o$ and annotation table Γ, its compilation version $[\![\mathcal{P}_o]\!]$ and its corresponding annotation table annotation table Γ', the following holds:

$$\mathsf{VCp}_{\mathcal{O}}(\mathcal{P}_o, \Gamma) = \mathsf{VCp}^{bc}_{\mathcal{O}}([\![\mathcal{P}_o]\!], \Gamma')$$

This theorem is sufficient to show the equality of the verification conditions, since the verification condition of both program due to the behavior subtyping are by construction equals.

The proof of the theorem is similar to the case of $\mathrm{JAVA}_{\mathcal{B}}$ and $\mathrm{JVM}_{\mathcal{B}}$. The only difference is for the case of the expressions, since the way of computing the WP at source level has really changed. To that end we prove the following lemma:

Lemma 13 *For all expression e compiled starting from position k, let $j = k + |[\![e]\!]|$ then for all postcondition ψ and number z such that*

$$\psi\{w_k \mapsto \mathbf{os}[k]\}_{k=0\ldots z} = \mathsf{wp}^{\mathsf{m}}_{l}(j)$$

we have

$$\mathsf{wp}_{\mathcal{O}}(k)(e, \psi)_{w_0}\{w_k \mapsto \mathbf{os}[k-1]\}_{k=1\ldots z} = \mathsf{wp}^{\mathsf{m}}_{l}(k)$$

4.3 Preservation of proof obligations for exceptions

4.3.1 Verification conditions for $\mathrm{JAVA}_{\mathcal{E}}$

Program specification for $\mathrm{JAVA}_{\mathcal{E}}$ programs

In order to characterize properly the normal and exceptional termination of programs, methods will be provided with a normal and exceptional postcondition for every possible exception type on which method execution may terminate. Thus, the specification tables are now extended with a new component: the exceptional postcondition of each methods.

An exceptional postcondition is a postcondition where the special variable exc binds the value of the raised expression, i.e. the exceptional reference. Since the semantics of a $\mathrm{JAVA}_{\mathcal{E}}$ program when an exception

is raised depend of the dynamic type of the exception, the specification language should be able to express this notion. Thus the syntax of proposition is extended with the following predicate: Which means that the dynamic type of the expression $\bar{e}$ in the heap H is a subclass of C.

Thus, a program $\mathcal{P}_e$ is provided with a specification table Γ where for every method m in every class C, Γ returns the already seen elements of method specification - the precondition Φ and the method normal postcondition Ψ. In addition, the specification table specifies for any exception type on which the method may terminate, the property that must hold in such an exceptional termination. This is done via the exceptional postcondition Ψ_{exc}. The postcondition Ψ_{exc} specifies depending on the type of the thrown exception object exc what is the respective property that must hold in the exceptional state, i.e. the Ψ_{exc} is of the form :

$$\begin{array}{l} \mathsf{typeof}(\mathsf{exc}) \preceq E_1 \Rightarrow \psi_1 \wedge \\ \mathsf{typeof}(\mathsf{exc}) \preceq E_2 \Rightarrow \psi_2 \wedge \\ \ldots \end{array}$$

We use the notation $\Psi_{exc}\{E \mapsto \psi\}$ as an abbreviation of the following :

$$\mathsf{typeof}(\mathsf{exc}) \preceq E_1 \Rightarrow \psi \wedge \neg(\mathsf{typeof}(\mathsf{exc}) \preceq E_1) \Rightarrow \Psi_{exc}$$

We write that the method signature m in class C is associated with the specification Φ, Ψ and Ψ_{exc} by the specification table Γ as follows $\Gamma(\mathcal{P}_e,\ \mathsf{m},\ \Phi,\ \Psi,\ \Psi_{exc})$.

Program and method validity Program validity has still the same meaning as in the previous sections

Definition 4.3.1 (Validity of programs in JAVA$_\mathcal{E}$**)** *A program $\mathcal{P}_e$ is valid if for every class C in $\mathcal{P}_e$ and for every method m declared in C, method m is valid in the sense of Def. 4.3.2.*

However, method validity should be adapted to the new operational semantics of the language. Given a method m with a specification Φ and Ψ (i.e. $\Gamma(\mathcal{P}_e, \mathsf{m}, \Phi, \Psi, \Psi_{exc})$), we say that m is valid w.r.t. its specification if a method starts execution in an initial local memory ρ and heap h, such that $(\rho,\ h) \models \Phi$ and terminates normally execution leaving a heap h' and returns a value v then the postcondition holds in the final state $(\rho,\ h), h', v \models \Psi$. Moreover, if the method starts execution in state $(\rho,\ h)$ such that $(\rho,\ h) \models \Phi$ and method terminates execution exceptionally by throwing an exceptional object r with a heap h' then the exceptional postcondition holds in the final state $(\rho,\ h), h', r \models \Psi_{exc}$. This is stated formally with the following definition:

Definition 4.3.2 (Validity of methods in JAVA$_\mathcal{E}$**)** *In program $\mathcal{P}_e$, a method m with specification Φ, Ψ, Ψ_{exc} is valid w.r.t. to this specification (notation $\models \Gamma(\mathcal{P}_e, \mathsf{m}, \Phi, \Psi, \Psi_{exc})$) if the following holds :*

$$\begin{array}{l} \forall \rho, h, h',\ [\mathsf{body}(\mathsf{m}), \rho, h] \Downarrow_\mathcal{E} (v,\ h') \Rightarrow \rho, h \models \Phi \Rightarrow \rho, h, h', v \models \Psi \wedge \\ \forall \rho, h, h',\ [\mathsf{body}(\mathsf{m}), \rho, h] \Downarrow_\mathcal{E} (r,\ h') \Rightarrow \rho, h \models \Phi \Rightarrow \rho, h, h', r \models \Psi_{exc} \end{array}$$

Verification condition generator for JAVA$_\mathcal{E}$

The definition of the verification condition generator for JAVA$_\mathcal{E}$ is shown in Fig. 4.6. The predicate transformer function $\mathsf{wp}_\mathcal{E}$ now takes not only the normal postcondition ψ but also an exceptional postcondition ψ_{exc}.

Actually, only the rules for expressions and statements which may throw or handle exceptions change. Thus, except the rules shown in the figure, the cases for the other language constructs remain the same as in the previous sections.

In these settings, for method invocation there are three cases - in case the receiver of the object is not null and the execution of the called method terminates normally, then the specified normal postcondition of the invoked method implies the normal ψ. In case that the receiver object is not null and the invocation

of the method terminates exceptionally, the specified exceptional postcondition of the called method must imply the exceptional postcondition of the invocation expression. Finally, if the receiver object is null, then the exceptional postcondition of the invocation expression must hold.

The try catch rule first calculates the precondition for the catch statement. The resulting precondition will update the exceptional postcondition ψ_{exc} for the exceptional type EType and upon the latter and the normal postcondition the precondition for the try statement will be calculated.

The verification conditions for a method w.r.t. its specification - precondition, normal postcondition and exceptional postcondition are the verification conditions over the method body and the formula which states that the specified precondition of the method implies the weakest precondition of the method body.

Finally, the verification conditions for a program are generated via the function $\mathsf{VCp}_{\mathcal{E}}$ which depends on $EvcC$ and $\mathsf{wp}_{\mathcal{E}}$ and is defined in the following way:

$$\mathsf{VCp}_{\mathcal{E}}(\mathcal{P}_e, \Gamma) = \cup_{\mathsf{C} \in \mathcal{P}_e} \mathsf{VCc}_{\mathcal{E}}(\mathsf{C}, \Gamma)$$

Theorem 4.3.3 (Soundness of the verification condition generator for $\text{JAVA}_{\mathcal{E}}$**)** *Let us have* $\text{JAVA}_{\mathcal{E}}$ *program* $\mathcal{P}_e$ *provided with specification table* Γ. *If the following conditions hold :*

1. *the verification conditions over methods are valid, i.e.* $\vdash \mathsf{VCp}_{\mathcal{E}}(\mathcal{P}_e, \Gamma)$
2. *overriding methods are proper behavioral subtypes*

then $\mathcal{P}_e$ *is valid in the sense of Definition 4.3.1*

4.3.2 Verification conditions for $\text{JVM}_{\mathcal{E}}$

The verification condition generator for the language is given in Fig. 4.7. The generation of verification conditions is based on the predicate transformers $\mathsf{wp}_i^{\mathcal{E}}$ and $\mathsf{wpE}^{\mathcal{E}}$ which are responsible for the normal postcondition and exceptional postconditions respectively. While the transformer $\mathsf{wp}_i^{\mathcal{E}}$ works in a similar way as the bytecode predicate transformers from the previous sections, the novel part here concerns $\mathsf{wpE}^{\mathcal{E}}$. The latter takes an index i of a bytecode instruction (we assume implicitly that the bytecode instructions are in the body of method m) and reconstructs the list of all the exception handlers in whose domain i is. The list of exception handlers $excH$ consists of pairs - the index at which the exception handler starts and the exception type that it manages. The first element in the list is the first element in the exception handler table which may handle exceptions from i, the second is the second such exception handler etc, i.e. the order in the list reflect which exception handler will be used to handle an exception thrown at index i. From the list $excH$ and the exceptional postcondition Ψ_{exc} of m , $\mathsf{wpE}^{\mathcal{E}}$ constructs the exceptional postcondition.

The verification conditions for behavioral subtyping remain the same as in $\text{JAVA}_{\mathcal{O}}$. We are now ready to state the soundness statement which is as follows:

Theorem 4.3.4 (Soundness of the verification condition generator for $\text{JVM}_{\mathcal{E}}$**)** *Let us have a* $\text{JVM}_{\mathcal{E}}$ *program* $\mathcal{P}_e$ *provided with specification table* Γ. *If the following conditions hold :*

1. *the verification conditions over methods are valid, i.e.* $\vdash \mathsf{VCp}_{\mathcal{O}}(\mathcal{P}_e, \Gamma)$
2. *overriding methods are proper behavioral subtypes, i.e.* $\vdash \mathsf{VCbs}_{\mathcal{E}}(\Gamma)$

then $\mathcal{P}_e$ *is valid in the sense of Definition 4.3.1*

4.3.3 Relation between the verification calculi for $\text{JAVA}_{\mathcal{E}}$ and $\text{JVM}_{\mathcal{E}}$

We shall establish few properties about the compiler which will play a role for the relation between the verification calculi of $\text{JAVA}_{\mathcal{E}}$ and $\text{JVM}_{\mathcal{E}}$ programs. In particular, we will show that the compiler will preserve the exception handlers, i.e. if a statement or expression in the source is protected by a particular exception handler, then the compilation or statement is protected by the compilation of the corresponding exception handler statement.

The next lemma concerns compilation of expressions. Because our programming language introduces exception handlers only via the try catch statement, this means that all the instructions resulting from an expression compilation for any exception thrown must have the same exception handler.

Lemma 14 (Exception handler compilation for expressions) *For every expression e and initial compilation index i the compiler returns the next free index j, the list of resulting* bytecode *instructions I and the the exception handler function* Handler*, i.e.* $j : [\![i, e]\!] = I, j, \mathsf{Handler}$ *such that :*

$$\forall\ k,\ \forall\ E, i \leq k \leq j \Rightarrow\ \mathsf{Handler}(k, E) = \mathsf{Handler}(j, E)$$

We shall turn now to the preservation of exception handlers in compilation of statements. Consider a try catch statement $\mathsf{try}\{s_1\}$ catch (EType) $\{s_2\}$. What happens there, is that every exception thrown by the try statement of type or subtype of EType will transfer the control to the catch statement s_2. Any other exception E thrown by the try statement will be processed in the same way in which E will be processed if thrown by the catch statement s_2. Thus, the compilation of a try catch statement modifies the exception handler function. On the other hand, from the fact that any other exception thrown by the try and catch statement is handled in the same way, implies that the exception handlers for any exception thrown by the last instructions in the compilation of the try and the catch substatements are the same.

Lemma 15 (Exception handler compilation for try catch statements) *Suppose that the statement s is of the form* $\mathsf{try}\{s_1\}$ $\mathsf{catch}(\mathsf{EType})\{s_2\}$ *and that its compilation starts at the initial index i and returns the next free index j, the list of resulting* bytecode *instructions I and the resulting exception handler function* $\mathsf{Handler}_1$*, i.e.* $[\![i, s]\!] = I, j, \mathsf{Handler}$*. More over, assume that the compilation of the substatement s_1 terminates at index k, i.e.* $[\![i, s_1]\!] = I_1, k, _$*. Then we have that*

- $\forall\ E,\ \neg E \preceq \mathsf{EType} \Rightarrow\ \mathsf{Handler}(k-1, E) = \mathsf{Handler}(j-1, E)$
- $\forall\ E,\quad E \preceq \mathsf{EType} \Rightarrow\ \mathsf{Handler}(k-1, E) = k+1$

In order to establish a similar property of the compiler for the rest of the statements of the language, we introduce the notion of direct of substatement: a statement s' is a direct substatement of s if $s' \subset s$, i.e. s' is a substatement of s and there is no other substatement s'', such that $s'' \subset s$ and $s' \subset s''$.

Lemma 16 (Exception handler compilation for statements not manipulating exceptions) *Suppose that the compilation of a non try catch statement s starts at the initial index i and the index j is the next free instruction index after the compilation, I the list of resulting bytecode instructions and* Handler *the resulting exception handler function, i.e.* $[\![i, s]\!] = I, j, \mathsf{Handler}$*. If s has a direct substatement s_1 whose compilation starts at index r and terminates leaving the next free index k for some exception handler* $[\![r, s_1]\!] = I, k, _$ *the following holds*

$$\forall\ E,\ \mathsf{Handler}(j-1, E) = \mathsf{Handler}(k-1, E)$$

We are now ready to state the relation between the verification conditions for the structured language $\text{JAVA}_{\mathcal{E}}$ and the bytecode language $\text{JVM}_{\mathcal{E}}$. As in the cases considered in the previous sections, this relation is a syntactic equivalence between the verification conditions of a $\text{JAVA}_{\mathcal{E}}$ program and the bytecode program produced by our compiler under the assumption bytecode that method specifications are the same and annotations are compiled in a proper way as discussed in Section 4.1.

Theorem 4.3.5 *Equivalence between* $\mathsf{VCp}_{\mathcal{E}}$ *and* $\mathsf{VCp}^{bc}_{\mathcal{E}}$ *Let us have the program $\mathcal{P}_e$ and its compilation $[\![\mathcal{P}_e]\!]$ and the annotation table Γ. Then the following holds:*

$$\mathsf{VCp}_{\mathcal{E}}(\mathcal{P}_e, \Gamma) = \mathsf{VCp}^{bc}_{\mathcal{E}}(\mathcal{P}_e, \Gamma)$$

To prove that equivalence in the current settings, we will need several more auxiliary properties. First, we must establish that the preconditions of a $\text{JAVA}_{\mathcal{E}}$ expression and its compilation are the same provided that that both the exceptional and normal postcondition are syntactically equivalent:

Lemma 17 *For the expression e and its compilation starting at index s and terminating with index i, a list of* bytecode *instructions I and exception handler* Handler*, i.e.* $[\![s, e]\!] = I, i, \mathsf{Handler}$*, the following holds:*

- *an assertion ψ and a natural number z such that* $\psi\{w_k \mapsto \mathsf{st}[k]\}_{k=1\ldots z}\{v \mapsto \mathsf{st}[0]\} = \mathsf{wp}^{\mathcal{E}}_l(i)$
- *an exceptional postcondition function ψ_e such that* $\psi_e = \mathsf{wpE}^{\mathcal{E}}(i-1)$

then we have

$$\mathsf{wp}_{\mathcal{E}}(e, \psi, \psi_e)_v\{w_k \mapsto \mathsf{st}[k]\}_{k=1\ldots z} = \mathsf{wp}^{\mathcal{E}}_l(s)$$

The proof is by induction over the expression structure. Actually the first case of the lemma coincides with the reasoning that we have used in Lemma 13. We shall now illustrate the proof for field access expressions.

- $e_1.\mathsf{f}$ By definition the weakest precondition for this case is as follows :

$$\begin{array}{l} \mathsf{wp}_{\mathcal{E}}(e_1.\mathsf{f}, \psi, \psi_{exc})_v = \\ \mathsf{wp}_{\mathcal{E}}(e_1, v_1 \neq \mathsf{null} \Rightarrow \psi\{v \mapsto \mathbf{h}(v_1.\mathsf{f})\} \wedge \\ \quad \forall h\ r, v_1 = \mathsf{null} \Rightarrow \mathsf{New}(\mathbf{h}, \mathsf{NullPntrExc}) = (h, r) \Rightarrow \psi_{exc}\{\mathbf{h}, \mathsf{exc} \mapsto h, r\}, \psi_{exc})_{v_1} \end{array}$$

  Remind that the compiler for field access expression is defined as follows:

$$[\![e_1.\mathsf{f}]\!] = s : [\![e_1]\!]; i : \mathsf{getfield}\ \mathsf{f}, i+1$$

  Looking at the definition of $\mathsf{wp}^{\mathcal{E}}_i$ for field access expressions in Fig.4.7 we can apply a similar reasoning to Lemma 13 and because of the hypothesis for the exceptional postcondition functions, we can establish that

$$\begin{array}{l} \mathsf{wp}^{\mathcal{E}}_i(i) = \\ (v_1 \neq \mathsf{null} \Rightarrow \psi \wedge \\ \ \forall h\ r, v_1 = \mathsf{null} \Rightarrow \mathsf{New}(\mathbf{h}, \mathsf{NullPntrExc}) = (h, r) \Rightarrow \\ \ \psi_{exc}\{\mathbf{h}, \mathsf{exc} \mapsto h, r\})\{v \mapsto \mathbf{h}(v_1.\mathsf{f})\}\{w_k \mapsto \mathsf{st}[k]\}_{k=1\ldots z}\{v_1 \mapsto \mathsf{st}[0]\} \end{array}$$

  In order, however, to apply the induction hypothesis over the expression e, we have to establish that the bytecode exceptional postcondition $\mathsf{wpE}^{\mathcal{E}}(i-1)$ for the instruction at index $i-1$ is the same as the source exception postcondition ψ_{exc}. For this we use Lemma 14 for the compilation of exception handlers, from which we can conclude that

$$\forall\ k,\ \forall\ E, s \leq k \leq i \Rightarrow\ \mathsf{Handler}_1(k, E) = \mathsf{Handler}_1(i, E)$$

  where k is the index at which the compilation starts and i is the last index of the compilation of $e_1.\mathsf{f}$. From this, the definition of $\mathsf{wpE}^{\mathcal{E}}$ and the initial hypothesis for exception postconditions we conclude that

$$\psi_{exc} = \mathsf{wpE}^{\mathcal{E}}(i-1)$$

  This allows us to apply the induction hypothesis over the subexpression e and this case is proved.

We need also to establish a lemma for preconditions of statements and their compilation.

Lemma 18 *Suppose that for the statement stmt starting at index s the compiler returns the sequence of instructions I, the next index k and the handler* Handler*, i.e.* $[\![s, stmt]\!] = I, k, \mathsf{Handler}$*. Assume that we have*

- *the annotation table Λ resulting from the annotation compilation in stmt*
- *an assertion ψ such that ψ does not contain stack subexpressions and moreover* $\mathsf{wp}^{\mathcal{E}}_l(k) = \psi$

- *an exceptional postcondition* ψ_e *such that* $\psi_{exc} = \mathsf{wpE}^{\mathcal{E}}(k-1)$

then if $\mathsf{wp}_{\mathcal{E}}(stmt, \psi) = \phi, \theta$ *then we have*

- $\phi = \mathsf{wp}_l^{\mathcal{E}}(s)$
- *for every proposition* ξ *in* θ*, there exists an index* i *such that* $s \leq i < k$ *and* $\Lambda(i) = A$ *such that* $\xi = A \Rightarrow \mathsf{wp}_l^{\mathcal{E}}(i)$

The proof is by induction over the statement structure. We sketch the first part of the proof for compositional and try catch statement. The second part follows directly by structural induction.

- $i_1; i_2$ The weakest precondition for the compositional statement in the presence of exception is as follows:

$$\begin{array}{l} \mathsf{wp}_{\mathcal{E}}(i_1; i_2, \psi, \psi_{exc}) = (\phi, \theta_1 \cup \theta_2), \textit{where} \\ \mathsf{wp}_{\mathcal{E}}(i_1, \psi', \psi_{exc}) = \phi, \theta_1 \\ \mathsf{wp}_{\mathcal{E}}(i_2, \psi, \psi_{exc}) = \psi', \theta_2 \end{array}$$

  and the compilation is as shown in the previous Section 2.3.3, Fig. 2.12:

$$[\![s : i_1; i_2]\!] = [\![i_1]\!]; r : [\![i_2]\!], k$$

  We can apply the induction hypothesis over statement i_2 and get that

$$\psi' = \mathsf{wp}_l^{\mathcal{E}}(r)$$

  We can establish that

$$\psi_{exc} = \mathsf{wpE}^{\mathcal{E}}(r-1)$$

  using the Lemma 16 from the previous subsection and the definition of $\mathsf{wpE}^{\mathcal{E}}$ in Fig. 4.7. We can thus apply the induction hypothesis over i_1, ψ' and ψ_{exc} and establish the case for compositional statements.

- $\mathsf{try}\{i_1\}$ catch (EType) $\{i_2\}$ The weakest precondition function for the try catch statement was defined as follows:

$$\begin{array}{l} \mathsf{wp}_{\mathcal{E}}(\mathsf{try}\{i_1\}\ \mathsf{catch}\ (\mathsf{EType})\ \{i_2\}, \psi, \psi_{exc}) = (\phi, \theta_1 \cup \theta_2), \textit{where} \\ \mathsf{wp}_{\mathcal{E}}(i_1, \psi, \psi_{exc}\{\mathsf{EType} \mapsto \psi'\}) = \phi, \theta_1 \\ \mathsf{wp}_{\mathcal{E}}(i_2, \psi, \psi_{exc}) = \psi', \theta_2 \end{array}$$

  Let us also remind the try catch statement compilation:

$$[\![\mathsf{try}\{i_1\}\ \mathsf{catch}\ (\mathsf{EType})\ \{i_2\}]\!] = (s : [\![i_1]\!]; l : \mathsf{goto}\ k;\ \ l+1 : [\![i_2]\!]; k :, \mathsf{Handler})$$

  We can apply the induction hypothesis over the catch statement i_2 and its compilation and obtain

$$\psi' = \mathsf{wp}_l^{\mathcal{E}}(l+1)$$

  From the initial hypothesis and the definition of the $\mathsf{wp}_l^{\mathcal{E}}$ for goto instruction, we obtain that:

$$\mathsf{wp}_l^{\mathcal{E}}(l) = \mathsf{wp}_l^{\mathcal{E}}(k)$$

  In order to apply the induction hypothesis over the statement i_1, we have also to establish that for every exception E the bytecode exceptional postcondition function $\mathsf{wpE}^{\mathcal{E}}$ returns the same predicate

for instruction at index $l-1$ as the source exceptional postcondition function $\psi_{exc}\{\mathsf{exc}, \mathsf{EType} \mapsto \psi'\}$. We can establish this using Lemma 15, i.e. we have that:

$$\psi_{exc}\{\mathsf{EType} \mapsto \psi'\} = \mathsf{wpE}^{\mathcal{E}}(l-1)$$

We can apply the induction hypothesis for this case and thus, obtain that

$$\mathsf{wp}_{\mathcal{E}}(i_1, \psi, \psi_{exc}\{\mathsf{EType} \mapsto \psi'\}) = \mathsf{wp}_l^{\mathcal{E}}(s)$$

which allows us to conclude that the lemma holds in this case

$$\frac{\begin{array}{c}\Gamma(\mathsf{m},\Phi,\Psi,\Psi_{exc}) \qquad \phi := \mathsf{wp}_{\mathcal{E}}(e_1, \mathsf{wp}_{\mathcal{E}}(e_2, \xi, \psi_{exc})_{v_2}, \psi_{exc})_{v_1} \\ \xi_1 := \Phi\{\mathsf{this}, \arg \mapsto v_1, v_2\} \\ \xi_2 := \forall h\ r, v_1 \neq \mathsf{null} \Rightarrow \Psi\{\mathsf{this}, \arg, \bar{\mathbf{h}}, \mathbf{h}, \mathsf{res} \mapsto v_1, v_2, \mathbf{h}, h, r\} \Rightarrow \psi\{\mathbf{h}, v \mapsto h, r\} \\ \xi_3 := \forall h\ r, v_1 \neq \mathsf{null} \Rightarrow \Psi_{exc}\{\mathsf{this}, \arg, \bar{\mathbf{h}}, \mathbf{h}, \mathsf{exc} \mapsto v_1, v_2, \mathbf{h}, h, r\} \Rightarrow \psi_{exc}\{\mathbf{h} \mapsto h\} \\ \xi_4 := \forall h\ r, v_1 = \mathsf{null} \Rightarrow \mathsf{New}(\mathbf{h}, \mathsf{NullPntrExc}) = (h, r) \Rightarrow \psi_{exc}\{\mathbf{h}, \mathsf{exc} \mapsto h, r\} \\ \xi := \xi_1 \wedge \xi_2 \wedge \xi_3 \wedge \xi_4\end{array}}{\mathsf{wp}_{\mathcal{E}}(e_1.\mathsf{m}(e_2), \psi, \psi_{exc})_v = \phi}$$

$$\frac{\begin{array}{c}\xi_1 := v_1 \neq \mathsf{null} \Rightarrow \psi\{v \mapsto \mathbf{h}(v_1.\mathsf{f})\} \\ \xi_2 := \forall h\ r, v_1 = \mathsf{null} \Rightarrow \mathsf{New}(\mathbf{h}, \mathsf{NullPntrExc}) = (h, r) \Rightarrow \psi_{exc}\{\mathbf{h}, \mathsf{exc} \mapsto h, r\} \\ \mathsf{wp}_{\mathcal{E}}(e_1, \xi_1 \wedge \xi_2, \psi_{exc})_{v_1} = \phi\end{array}}{\mathsf{wp}_{\mathcal{E}}(e_1.\mathsf{f}, \psi, \psi_{exc})_v = \phi}$$

$$\frac{\begin{array}{c}\xi_1 := v_1 \neq \mathsf{null} \Rightarrow \psi\{\mathbf{h} \mapsto \mathbf{h}\{v_1.\mathsf{f} \mapsto v_2\}\} \\ \xi_2 := \forall h\ r, v_1 = \mathsf{null} \Rightarrow \mathsf{New}(\mathbf{h}, \mathsf{NullPntrExc}) = (h, r) \Rightarrow \psi_{exc}\{\mathbf{h}, \mathsf{exc} \mapsto h, r\} \\ \mathsf{wp}_{\mathcal{E}}(e_1, \mathsf{wp}_{\mathcal{E}}(e_2, \xi_1 \wedge \xi_2, \psi_{exc})_{v_2}, \psi_{exc})_{v_1} = \phi\end{array}}{\mathsf{wp}_{\mathcal{E}}(e_1.\mathsf{f} := e_2, \psi, \psi_{exc}) = (\phi, \emptyset)}$$

$$\frac{\mathsf{wp}_{\mathcal{E}}(i_1, \psi', \psi_{exc}) = \phi, \theta_1 \quad \mathsf{wp}_{\mathcal{E}}(i_2, \psi, \psi_{exc}) = \psi', \theta_2}{\mathsf{wp}_{\mathcal{E}}(i_1; i_2, \psi, \psi_{exc}) = (\phi, \theta_1 \cup \theta_2)}$$

$$\frac{\mathsf{wp}_{\mathcal{E}}(i_1, \psi, \psi_{exc}\{\mathsf{EType} \mapsto \psi'\}) = \phi, \theta_1 \quad \mathsf{wp}_{\mathcal{E}}(i_2, \psi, \psi_{exc}) = \psi', \theta_2}{\mathsf{wp}_{\mathcal{E}}(\mathsf{try}\{i_1\}\ \mathsf{catch}\ (\mathsf{EType})\ \{i_2\}, \psi, \psi_{exc}) = (\phi, \theta_1 \cup \theta_2)}$$

$$\frac{\begin{array}{c}\xi_1 := v \neq \mathsf{null} \Rightarrow \psi_{exc}\{\mathsf{exc} \mapsto v\} \\ \xi_2 := \forall h\ r, v = \mathsf{null} \Rightarrow \mathsf{New}(\mathbf{h}, \mathsf{NullPntrExc}) = (h, r) \Rightarrow \psi_{exc}\{\mathbf{h}, \mathsf{exc} \mapsto h, r\} \\ \mathsf{wp}_{\mathcal{E}}(e, \xi_1 \wedge \xi_2, \psi_{exc})_v = \psi'\end{array}}{\mathsf{wp}_{\mathcal{E}}(\mathsf{throw}\ e, \psi, \psi_{exc}) = (\psi', \emptyset)}$$

$$\frac{\mathsf{wp}_{\mathcal{E}}(\mathsf{body}(\mathsf{m}), \Psi, \Psi_{exc}) = \phi, \theta}{\mathsf{VCgen}_{\mathcal{E}}(\mathsf{m}, \Phi, \Psi, \Psi_{exc}) = \{\Phi \Rightarrow \phi\} \cup \theta}$$

$$\frac{\Gamma(\mathsf{m}, \Phi, \Psi, \Psi_{exc})}{\mathsf{VCc}_{\mathcal{E}}(\mathsf{C}, \Gamma) = \cup_{\mathsf{m} \in \mathsf{C}} \mathsf{VCgen}_{\mathcal{E}}(\mathsf{m}, \Phi, \Psi, \Psi_{exc})}$$

Figure 4.6: WEAKEST PRECONDITION FOR THE EXTENSIONS IN $\text{JAVA}_{\mathcal{E}}$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invoke\ m} \\ \Gamma(\mathsf{m}, \Phi, \Psi, \Psi_{exc}) \qquad \xi := \mathsf{wp}^{\mathcal{E}}_l(i+1) \\ \phi := \Phi\{\bar{\mathsf{this}}, \bar{arg}, \bar{\mathbf{h}} \mapsto \mathsf{st}[1], \mathsf{st}[0], \mathbf{h}\} \wedge \phi_n \wedge \phi_{ec} \wedge \phi_{eu} \\ \phi_n := \mathsf{st}[1] \neq \mathsf{null} \Rightarrow \forall h\ r, \Psi\{\bar{\mathsf{this}}, \bar{arg}, \mathsf{res}, \bar{\mathbf{h}}, \mathbf{h} \mapsto \mathsf{st}[1], \mathsf{st}[0], r, \mathbf{h}, h\} \Rightarrow \xi\{\mathsf{st}, \mathbf{h} \mapsto r ::\uparrow \mathsf{st}, h\} \\ \phi_{ec} := \forall h\ r, \mathsf{st}[1] \neq \mathsf{null} \Rightarrow \Psi_{exc}\{\bar{\mathsf{this}}, \bar{arg}, \bar{\mathbf{h}}, \mathbf{h}, \mathsf{exc} \mapsto \mathsf{st}[1], \mathsf{st}[0], \mathbf{h}, h, r\} \Rightarrow \mathsf{wpE}^{\mathcal{E}}(i)\{\mathbf{h} \mapsto h\} \\ \phi_{eu} := \forall h\ e, \mathsf{st}[1] = \mathsf{null} \Rightarrow \mathsf{New}(H, \mathsf{NullPntrExc}) = (h, e) \Rightarrow \mathsf{wpE}^{\mathcal{E}}(i)\{\mathbf{h}, \mathsf{exc} \mapsto h, e\} \end{array}}{\mathsf{wp}^{\mathcal{E}}_i(i) = \phi}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{throw} \\ \xi_1 := \mathsf{st}[0] \neq \mathsf{null} \Rightarrow \psi_{exc}\{\mathsf{exc} \mapsto \mathsf{st}[0]\} \\ \xi_2 := \forall h\ r, \mathsf{st}[0] = \mathsf{null} \Rightarrow \mathsf{New}(\mathbf{h}, \mathsf{NullPntrExc}) = (h, r) \Rightarrow \mathsf{wpE}^{\mathcal{E}}(i)\{\mathbf{h}, \mathsf{exc} \mapsto h, r\} \end{array}}{\mathsf{wp}^{\mathcal{E}}_i(i) = \xi_1 \wedge \xi_2}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{putfield\ f} \quad \phi := \mathsf{wp}^{\mathcal{E}}_l(i+1) \\ \xi_1 := \mathsf{st}[1] \neq \mathsf{null} \Rightarrow \phi\{\mathbf{h}, \mathsf{st} \mapsto \mathbf{h}\{\mathsf{st}[1].\mathsf{f} \mapsto \mathsf{st}[0]\}, \uparrow^2 \mathsf{st}\} \\ \xi_2 := \forall h\ r, \mathsf{st}[1] = \mathsf{null} \Rightarrow \mathsf{New}(H, \mathsf{NullPntrExc}) = (h, r) \Rightarrow \mathsf{wpE}^{\mathcal{E}}(i)\{\mathbf{h}, \mathsf{exc} \mapsto h, r\} \end{array}}{\mathsf{wp}^{\mathcal{E}}_i(i) = \xi_1 \wedge \xi_2}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{getfield\ f} \quad \phi := \mathsf{wp}^{\mathcal{E}}_l(i+1) \\ \xi_1 := \mathsf{st}[1] \neq \mathsf{null} \Rightarrow \phi\{\mathsf{st} \mapsto \mathbf{h}(\mathsf{st}[0].\mathsf{f}) ::\uparrow \mathsf{st}\} \\ \xi_2 := \forall h\ r, \mathsf{st}[1] = \mathsf{null} \Rightarrow \mathsf{New}(H, \mathsf{NullPntrExc}) = (h, r) \Rightarrow \mathsf{wpE}^{\mathcal{E}}(i)\{\mathbf{h}, \mathsf{exc} \mapsto h, r\} \end{array}}{\mathsf{wp}^{\mathcal{E}}_i(i) = \xi_1 \wedge \xi_2}$$

$$\frac{\begin{array}{l} \mathsf{Handlers}(i) = excH \\ \mathsf{wpE}^{\mathcal{E}}(i, l) = \begin{cases} \Psi_{exc} & \mathit{if}\ excH = nil \\ \mathsf{wpE}^{\mathcal{E}}(i, t)\{E \mapsto \mathsf{wp}^{\mathcal{E}}_i(j)\} & \mathit{if}\ excH = (j, E)::t \end{cases} \end{array}}{\mathsf{wpE}^{\mathcal{E}}(i) = \xi}$$

$$\frac{\Gamma(\mathsf{m}, \Phi, \Psi, \Psi_{exc}) \quad \mathsf{body}(\mathsf{m}) = \dot{\mathcal{P}}}{\mathsf{VCgen}^{bc}_{\mathcal{E}}(\mathsf{m}, \Phi, \Psi, \Psi_{exc}) = \{\psi \Rightarrow \mathsf{wp}^{\mathcal{E}}_i(\dot{\mathcal{P}}[i]) \mid \Lambda(i) = \psi\} \cup \{\Phi \Rightarrow \mathsf{wp}^{\mathcal{E}}_i(\dot{\mathcal{P}}[0])\}}$$

$$\frac{\Gamma(\mathsf{m}, \Phi, \Psi, \Psi_{exc})}{\mathsf{VCc}^{bc}_{\mathcal{E}}(\mathsf{C}, \Gamma) = \cup_{\mathsf{m} \in \mathsf{C}} \mathsf{VCgen}^{bc}_{\mathcal{E}}(\mathsf{m}, \Phi, \Psi, \Psi_{exc})}$$

Figure 4.7: Verification condition generator for $\mathrm{JVM}_{\mathcal{E}}$

Chapter 5

Proof-transforming compilation for programs with abrupt termination

In the last chapters, we have seen how proofs can be produced in such a way that the transformation of them is simple, using two VCGens that produce structurally the same verification conditions. Instead of using a verification condition generator we now look at a way to transform proofs using a Hoare logic for source and bytecode. In this case, the proof contains the full structure of the method bodies, which makes the translation more complicated. Knowing that we can automatically transform proofs directly into a Hoare proof on bytecode opens the possibility to use a proof-transforming compiler to directly prove properties on the MOBIUS base logic.

If the source and target languages are close, the proof translation is still simple. However, if they are not close and the compilation function is complex, the translation can be hard. For example, proof-transformation from a subset of Java with `try-catch`, `try-finally` and `break` statements to Java bytecode is difficult. Compiling these statements in isolation is simple, but the compilation of their interplay is not. In the earlier chapters, we left away this difficulty of abrupt termination. Now we extend the language subset to show interesting aspects of abrupt termination.

A `try-finally` statement is compiled using *code duplication*: the `finally` block is put after the `try` block. If `try-finally` statements are used inside of a `while` loop, the compilation of `break` statements first duplicates the `finally` blocks and then inserts a jump to the end of the loop. Furthermore, the generation of exception tables is also harder. The code duplicated before the `break` may have exception handlers different from those of the enclosing `try` block. Therefore, the exception table must be changed so that exceptions are caught by the appropriate handlers. In this chapter, we present the first Proof-Transforming Compiler that handles these complications.

Outline. The source language and its Hoare-style logic are introduced in Section 5.1. We present the bytecode language and its logic in Section 5.2. In Section 5.3, we define the proof transformation. Section 5.4 illustrates proof transformations by an example. Section 5.5 states a soundness theorem.

5.1 Source language and logic

The source language we consider is similar to $\text{JAVA}_{\mathcal{E}}$ (see Section 2.3). We leave away all language constructs that would be simple to transform and add the additional statement `break` as well as the `finally`-clause to deal with abrupt termination. Its definition is the following:

$$\begin{array}{lcl} \mathit{exp} & ::= & \mathit{literal} \mid \mathit{var} \mid \mathit{exp}\ \mathit{op}\ \mathit{exp} \\ \mathit{stm} & ::= & x = \mathit{exp} \mid \mathit{stm}; \mathit{stm} \mid \texttt{while}\ (\mathit{exp})\ \mathit{stm} \mid \texttt{break}\ ; \mid \texttt{if}\ (\mathit{exp})\ \mathit{stm}\ \texttt{else}\ \mathit{stm} \\ & \mid & \texttt{try}\ \mathit{stm}\ \texttt{catch}\ (\mathit{type}\ \mathit{var})\ \mathit{stm} \mid \texttt{try}\ \mathit{stm}\ \texttt{finally}\ \mathit{stm} \mid \texttt{throw}\ \mathit{exp}\ ; \end{array}$$

To avoid `return` statements, we assume that the return value of every method is assigned to a special local variable named **result** (this is the only discordance with respect to Java). Moreover, we assume that the expressions are side-effect-free and cannot throw exceptions.

The subset of Java is small, but the combination of `while`, `breaks`, `try-catch` and `try-finally` statements produces an interesting subset especially from the point of view of compilation. The code duplication used by the compiler for `try-finally` statements increases the complexity of the compilation and translation functions, specially the formalization and its soundness proof.

In our technical report [27], the source languages also includes object-oriented features such as cast, new, read and write field, and method invocation. In this chapter, we only present the most interesting features.

5.1.1 Method and statement specifications

The logic is based on the programming logic introduced in [26, 34, 35]. We have modified it and proposed new rules for `while` including `break` and exceptions, `try-catch` and `try-finally`. In [35], a special variable χ is used to capture the status of the program such as normal or exceptional status. This variable is not necessary in the bytecode proof since non-linear control flow is implemented via jumps. To eliminate the χ variable, we use Hoare triples with two or three postconditions to encode the status of the program execution. This simplifies not only the translation but also the presentation.

Properties of methods are expressed by Hoare triples of the form $\{P\}\ \ T.m\ \ \{\ Q_n\ ,\ Q_e\ \}$, where $P,\ Q_n,\ Q_e$ are first-order formulas and $T.m$ is a method m declared in class T. The third component of the triple consists of a normal postcondition (Q_n), and an exceptional postcondition (Q_e). We call such a triple *method specification*.

Properties of statements are specified by Hoare triples of the form $\{P\}\ \ S\ \ \{Q_n, Q_b, Q_e\}$, where P, Q_n, Q_b, Q_e are first-order formulas and S is a statement. For statements, we have a normal postcondition (Q_n), a postcondition after the execution of a `break` (Q_b), and an exceptional postcondition (Q_e).

The triple $\{P\}\ \ S\ \ \{Q_n, Q_b, Q_e\}$ defines the following refined partial correctness property: if S's execution starts in a state satisfying P, then (1) S terminates normally in a state where Q_n holds, or S executes a `break` statement and Q_b holds, or S throws an exception and Q_e holds, or (2) S aborts due to errors or actions that are beyond the semantics of the programming language, e.g., memory allocation problems, or (3) S runs forever.

5.1.2 Rules

Figure 5.1 shows the rules for compositional, `while`, `break`, `try-catch`, and `throw` statements. In the compositional statement, the statement s_1 is executed first. The statement s_2 is executed if and only if s_1 has terminated normally.

In the `while` rule, the execution of the statement s_1 can produce three results: either (1) s_1 terminates normally and I holds, or (2) s_1 executes a `break` statement and Q_b holds, or (3) s_1 throws an exception and R_e holds. The postcondition of the `while` statement expresses that either the loop terminates normally and $(I \wedge \neg e) \vee\ Q_b$ holds or throws an exception and R_e holds. The `break` postcondition is false, because after a `break` within the loop, execution continues normally after the loop.

The `break` rule sets the normal and exception postcondition to false and the `break` postcondition to P due to the execution of a `break` statement.

In the `try-catch` rule, the execution of the statement s_1 can produce three different results: (1) s_1 terminates normally and Q_n holds or terminates with a `break` and Q_b holds. In these cases, the statement s_2 is not executed and the postcondition of the `try-catch` is the postcondition of s_1; (2) s_1 throws an exception and the exception is not caught. The statement s_2 is not executed and the `try-catch` finishes in an exception mode. The postcondition is $Q_e'' \ \wedge\ \tau(excV) \npreceq T$, where τ yields the runtime type of an object, $excV$ is a variable that stores the current exception, and $\preceq$ denotes subtyping; (3) s_1 throws an exception and the exception is caught. In the postcondition of s_1, $Q_e' \ \wedge\ \tau(excV) \preceq T$ specifies that the exception is caught. Finally, s_2 is executed producing the postcondition. Note that the postcondition is not only a normal postcondition: it also has to take into account that s_2 can throw an exception or can execute a `break`.

Similar to `break`, the `throw` rule modifies the postcondition P by updating the exception component of the state with the just evaluated reference.

compositional

$$\frac{\{P\}\ \ s_1\ \ \{Q_n, R_b, R_e\} \qquad \{Q_n\}\ \ s_2\ \ \{R_n, R_b, R_e\}}{\{P\}\ \ s_1; s_2\ \ \{R_n, R_b, R_e\}}$$

while

$$\frac{\{e\ \wedge\ I\}\ \ s_1\ \ \{I, Q_b, R_e\}}{\{I\}\ \ \texttt{while}\ (e)\ s_1\ \ \{((I \wedge \neg e) \vee\ Q_b), \mathit{false}, R_e\}}$$

break

$$\frac{}{\{P\}\ \ \texttt{break}\ \ \{\mathit{false}, P, \mathit{false}\}}$$

try-catch

$$\frac{\{P\}\ \ s_1\ \ \{Q_n, Q_b, Q\} \qquad \{Q'_e[e/\mathit{excV}]\}\ \ s_2\ \ \{Q_n, Q_b, R_e\}}{\{P\}\ \ \texttt{try}\ s_1\ \texttt{catch}\ (T\ \ e)\ s_2\ \ \{Q_n, Q_b, R\}}$$

where

$$Q \equiv (\ (Q''_e\ \wedge\ \tau(\mathit{excV}) \not\preceq T) \vee (Q'_e\ \wedge\ \tau(\mathit{excV}) \preceq T)\ )$$

$$R \equiv (R_e\ \vee\ (Q''_e\ \wedge\ \tau(\mathit{excV}) \not\preceq T)\ )$$

throw

$$\frac{}{\{P[e/\mathit{excV}]\}\ \ \texttt{throw}\ e\ \ \{\mathit{false}, \mathit{false}, P\}}$$

Figure 5.1: Rules for composition, `while`, `break`, `try-catch`, and `throw`.

To define the rule for `try-finally`, we have to treat a special case, illustrated through the example in Figure 5.2.

```
void foo () {
   int b = 1;
   while (true) {
      try { throw new Exception(); }
      finally { b++; break; }
   }
   b++;
}
```

Figure 5.2: The exception raised in the `try` block is not handled, yet the method terminates normally.

The exception thrown in the `try` block is never caught. However, the loop terminates normally due to the execution of the `break` statement in the `finally` block. Thus, the value of b at the end of *foo* is 3.

If an exception occurs in a `try` block, it will be re-raised after the execution of the `finally` block. If both the `try` and the `finally` block throw an exception, the latter takes precedence. The following table summarizes the status of the program after the execution of the `try-finally`:

		finally		
		normal	**break**	**exc$_2$**
try	**normal**	normal	break	exc_2
	break	break	break	exc_2
	exc$_1$	exc_1	break	exc_2

We use the fresh variable $eTmp$ to store the exception occurred in s_1 because another exception might be raised and caught in s_2. In this case, we still need to have access to the first exception of s_1 because this exception is the result of that statement [35]. We use the fresh variable $\mathcal{X}Tmp$ to store the status of the program after the execution of s_1. The possible values of $\mathcal{X}Tmp$ are: *normal*, *break*, and *exc*. Depending on the status after the execution of s_2, we need to propagate an exception or change the status of the program to *break*. The rule is the following:

$$\frac{\begin{array}{c}\{P\}\ \ s_1\ \ \{Q_n, Q_b, Q_e\}\\ \{Q\}\ \ s_2\ \ \{R, R'_b, R'_e\}\end{array}}{\{P\}\ \ \texttt{try}\ s_1\ \texttt{finally}\ s_2\ \ \{R'_n, R'_b, R'_e\}}$$

where

$$\begin{array}{l} Q \equiv \left(\begin{array}{l} (Q_n \wedge \mathcal{X}Tmp = normal)\ \vee (Q_b \wedge \mathcal{X}Tmp = break) \vee \\ (\ \ Q_e[eTmp/excV] \wedge \mathcal{X}Tmp = exc \wedge eTmp = excV\ \ ) \end{array} \right) \\ R \equiv \left(\begin{array}{l} (R'_n \wedge \mathcal{X}Tmp = normal)\ \vee\ (R'_b \wedge \mathcal{X}Tmp = break) \vee \\ (R'_e \wedge \mathcal{X}Tmp = exc) \end{array} \right) \end{array}$$

Furthermore, the logic contains language-independent rules such as the rule of consequence (see [34]).

5.2 Bytecode language and logic

The JVM$_\mathcal{E}$ instructions used to compile the source language are: push v, load x, store x, binop op, goto l, and throw. We use an additional instruction iftrue l which transfers control to the point l if the topmost element of the stack is true and unconditionally pops it. This instruction is not defined in JVM$_\mathcal{E}$ but it is a shortcut for the JVM$_\mathcal{E}$ instruction sequence 'push true; if $= l$'. As you can see, we assume that the bytecode language has a type boolean in order to concentrate on abrupt termination.

The bytecode logic is a Hoare-style program logic which allows one to formally verify that implementations satisfy interface specifications given as pre- and postconditions. We use the bytecode logic developed by Bannwart and Müller [4]. As part of Task 3.1, we have shown that the MOBIUS base logic covers the chosen logic, that is, for each rule of the Bannwart-Müller Logic an interpretation can be given that is derivable in the MOBIUS base logic.

5.2.1 Method and Instruction Specifications

To make proof transformation feasible, it is essential that the source logic and the bytecode logic are similar in their structure. In particular, they treat methods in the same way, they contain the same language-independent rules, and triples have a similar meaning.

Analogously to the source logic, properties of methods are expressed by method specifications of the form form {P} $T.mp$ $\{Q_n,\ Q_e\}$. Properties of method bodies are expressed by Hoare triples of the form {P} *comp* {Q}, where P, Q are first-order formulas and *comp* is a method body. The triple {P} *comp* {Q} expresses the following refined partial correctness property: if the execution of *comp* starts in a state satisfying P, then (1) *comp* terminates in a state where Q holds, or (2) *comp* aborts due to errors or actions that are beyond the semantics of the programming language, or (3) *comp* runs forever.

The unstructured control flow of bytecode programs makes it difficult to handle instruction sequences, because jumps can transfer control into and from the middle of a sequence. Therefore, the logic treats

each instruction individually: each individual instruction I_l in a method body p has a precondition E_l. An instruction with its precondition is called an *instruction specification*, written as $\{E_l\}\ l : I_l$.

The meaning of an instruction specification $\{E_l\}\ l : I_l$ cannot be defined in isolation. $\{E_l\}\ l : I_l$ expresses that if the precondition E_l holds when the program counter is at position l, the precondition $E_{l'}$ of I_l's successor instruction I'_l holds after normal termination of I_l.

5.2.2 Rules

All the rules for instructions, except for method calls, have the following form:

$$\frac{E_l \Rightarrow wp_p^1(I_l)}{\mathrm{A} \vdash \{E_l\}\ l : I_l}$$

where $wp_p^1(I_l)$ denotes the *local weakest precondition* of instruction I_l. Such a rule specifies that the precondition of I_l has to imply the weakest precondition of I_l with respect to all possible successor instructions of I_l. The definition of wp_p^1 is shown in Table 5.1.

Within an assertion, the current stack is referred to as s and its elements are denoted by non-negative integers: element 0 is the topmost element, etc. The interpretation $[E_l] : State \times Stack \rightarrow Value$ for s is

$$\begin{array}{ll} [s(0)]\langle S, (\sigma, v)\rangle & = v \quad \text{and} \\ [s(i+1)]\langle S, (\sigma, v)\rangle & = [s(i)]\langle S, \sigma\rangle \end{array}$$

The functions *shift* and *unshift* define the substitutions that occur when values are pushed onto and popped from the stack, respectively:

$$\begin{array}{ll} shift(E) & = E[s(i+1)/s(i) \mid \forall i \in \mathbb{N}] \\ unshift & = shift^{-1} \end{array}$$

I_l	$wp_p^1(I_l)$
push v	$unshift(E_{l+1}[v/s(0)])$
load x	$unshift(E_{l+1}[x/s(0)])$
store x	$(shift(E_{l+1}))[s(0)/x]$
binop op	$(shift(E_{l+1}))[s(1)\ op\ s(0)/s(1)]$
goto l'	$E_{l'}$
iftrue l'	$(\neg s(0) \Rightarrow shift(E_{l+1})) \wedge (s(0) \Rightarrow shift(E_{l'}))$

Table 5.1: Definition of function wp_p^1.

5.3 Proof Translation

Our proof-transforming compiler is based on two *transformation functions*, ∇_S and ∇_E, for statements and expressions, respectively. Both functions yield a sequence of bytecode instructions and their specification. The Proof-Transforming Compiler takes a list of classes with their proofs and returns the bytecode classes with their proofs.

The function ∇_E generates a bytecode proof from a source expression and a precondition for its evaluation. The function ∇_S generates a bytecode proof and an exception table from a source proof. These functions are defined as a composition of the translations of its sub-trees. The signatures are the following:

$$\nabla_E \ : Precondition \times Expression \times Postcondition \times Label \rightarrow BytecodeProof$$

$$\nabla_S \ : ProofTree \times Label \times Label \times Label \times List[Finally] \times ExcTable \rightarrow [BytecodeProof \times ExcTable]$$

Type	Typical use
$Precondition \cup Postcondition$	P, Q, R, U, V
$ProofTree$ (for source language only)	$T_{S_1}, T_{S_2}, Tree_i$
$ProofTree$ (for `finally` only)	T_{F_i}
$List[Finally]$	f
$ExceptionTable$	et_i
$ExceptionTable$ (for `finally` only)	et'_i
$BytecodeProof$	B_{S_1}, B_{S_2}
$InstrSpec$	$b_{\mathsf{push}}, ..., b_{\mathsf{iftrue}}$
$Label$	$l_{start}, l_{next}, l_{break}$, $l_b, l_c, ..., l_g$

Table 5.2: Naming conventions.

In ∇_E, the label is used as the starting label of the translation. *ProofTree* is a derivation in the source logic. In ∇_S, the three labels are: (1) l_{start} for the first label of the resulting bytecode; (2) l_{next} for the label after the resulting bytecode; this is for instance used in the translation of an `else` branch to determine where to jump at the end; (3) l_{break} for the jump target for `break` statements.

The *BytecodeProof* type is defined as a list of *InstrSpec*, where *InstrSpec* is an instruction specification. The *Finally* type, used to translate `finally` statements, is defined as a tuple [*ProofTree*, *ExcTable*]. Furthermore, the ∇_S takes an exception table as parameter and produces an exception table. This is necessary because the translation of `break` statements can lead to a modification of the exception table as described above. (more details are presented in Section 5.3.3).

The *ExcTable* type is defined as follows:

$$\begin{array}{ll} ExcTable & := List[ExcTableEntry] \\ ExcTableEntry & := [Label, Label, Label, Type] \end{array}$$

In the *ExcTableEntry* type, the first label is the *starting label* of the exception line, the second denotes the *ending label*, and the third is the *target label*. An exception of type T_1 thrown at line l is caught by the exception entry $[l_{start}, l_{end}, l_{targ}, T_2]$ if and only if $l_{start} \leq l < l_{end}$ and $T_1 \preceq T_2$. Control is then transferred to l_{targ}.

In the following, we present the proof translation for compositional rule, `while`, `try-finally`, and `break`. Table 5.2 comprises the naming conventions we use in the rest of this chapter.

5.3.1 Compositional Statement

Let T_{S_1} and T_{S_2} be the following proof trees:

$$T_{S_1} \equiv \frac{Tree_1}{\{P\} \quad s_1 \quad \{Q_n, R_b, R_e\}}$$

$$T_{S_2} \equiv \frac{Tree_2}{\{Q_n\} \quad s_2 \quad \{R_n, R_b, R_e\}}$$

$$T_{S1;S2} \equiv \frac{T_{S_1} \qquad\qquad T_{S_2}}{\{P\} \quad s_1; s_2 \quad \{R_n, R_b, R_e\}}$$

In the translation of T_{S_1}, the label l_{next} is the start label of the translation of s_2, say l_b. The translation of T_{S_2} uses the exception table produced by the translation of T_{S_1}, et_1. The translation of $T_{S1;S2}$ yields

the concatenation of the bytecode proofs for the sub-statements and the exception table produced by the translation of T_{S_2}.

Let $[B_{S_1}, et_1]$ and $[B_{S_2}, et_2]$ be of type $[BytecodeProof, ExcTable]$:

$$[B_{S_1}, et_1] = \nabla_S\ (T_{S_1}, l_{start}, l_b, l_{break}, f, et)$$
$$[B_{S_2}, et_2] = \nabla_S\ (T_{S_2}, l_b, l_{next}, l_{break}, f, et_1)$$

The translation is defined as follows:

$$\nabla_S\ (\ T_{S1;S2},\ \ l_{start}, l_{next}, l_{break}, f,\ \ et) = [B_{S_1} + B_{S_2}\ ,\ \ et_2]$$

The bytecode for s_1 establishes Q_n, which is the precondition of the first instruction of the bytecode for s_2. Therefore, the concatenation $B_{S_1} + B_{S_2}$ produces a sequence of valid instruction specifications. We will formalize soundness in Section 5.5.

5.3.2 While Statement

Let T_{S_1} and T_{while} be the following proof trees:

$$T_{S_1} \equiv \frac{Tree_1}{\{e\ \wedge\ I\}\ \ s_1\ \ \{I, Q_b, R_e\}}$$

$$T_{while} \equiv \frac{T_{S_1}}{\{I\}\ \ \texttt{while}\ (e)\ s_1\ \ \{(I \wedge \neg e) \vee\ \ Q_b, false, R_e\}}$$

In this translation, first the loop expression is evaluated at l_c. If it is true, control is transferred to l_b, the start label of the loop body. In the translation of T_{S_1}ing, the start label and next labels are l_b and l_c. The break label is the end of the loop (l_{next}). Furthermore, the finally list is set to $\emptyset$, because a `break` inside the loop jumps to the end of the loop without executing any `finally` blocks.

Let b_{goto} and b_{iftrue} be instruction specifications and B_{S_1} and B_e be bytecode proofs:

$$\begin{aligned} b_{\mathsf{goto}} = &\quad \{I\} \quad l_a : \mathsf{goto}\ l_c \\ [B_{S_1}, et_1] = &\quad \nabla_S\ (T_{S_1},\ \ l_b, l_c, l_{next},\ \ \emptyset, et) \\ B_e = &\quad \nabla_E\ (\ I,\ \ e,\ \ (shift(I)\ \wedge\ s(0) = e)\ ,\ \ l_c\ ) \\ b_{\mathsf{iftrue}} = &\quad \{shift(I)\ \wedge\ s(0) = e\} \quad l_d :\ \mathsf{iftrue}\ l_b \end{aligned}$$

The definition of the translation is the following:

$$\nabla_S\ (T_{while},\ \ l_{start}, l_{next}, l_{break},\ f,\ \ et) = [\ b_{\mathsf{goto}} + B_{S_1} + B_e + b_{\mathsf{iftrue}}\ ,\ \ et_1\ ]$$

The instruction b_{goto} establishes I, which is the precondition of the successor instruction (the first instruction of B_e). B_e establishes $shift(I)\ \wedge\ s(0) = e$ because the evaluation of the expression pushes the result on top of the stack. This postcondition implies the precondition of the successor instruction b_{iftrue}. b_{iftrue} establishes the preconditions of both possible successor instructions, namely $e\ \wedge\ I$ for the successor l_b (the first instruction of B_{S_1}), and $I\ \wedge\ \neg e$ for l_{next}. Finally, B_{S_1} establishes I, which implies the precondition of its successor B_e, $(I \wedge \neg e) \vee\ Q_b$. Therefore, the produced bytecode proof is valid.

5.3.3 Try-Finally Statement

Sun's newer Java compilers translate `try-finally` statements using code duplication. Consider the following example:

```
while (i < 20) {
    try {
        try {
            try { ... break; ... }
            catch (Exception e) { i = 9; }
        }
        finally { throw new Exception(); }
    }
    catch (Exception e) { i = 99; }
}
```

The `finally` body is duplicated before the `break`. But the exception thrown in the `finally` bock must be caught by the outer `try-catch`. To achieve that, the compiler creates, in the following order, exception lines for the outer `try-catch`, for the `try-finally`, and for the inner `try-catch`. When the compiler reaches the `break`, it divides the exception entry of the inner `try-catch` and `try-finally` into two parts so that the exception is caught by the outer `try-finally`. To be able to divide the exception table the compiler needs to compare the exception entries. This is why our *Finally* type consists of a proof tree (for the duplicated code) and an exception table. Note that we have a list of *Finally* to handle nested `try-finally` statements.

Let T_{S_1}, T_{S_2} and $T_{try-finally}$ be the following proof trees:

$$T_{S_1} \equiv \frac{Tree_1}{\{P\} \quad s_1 \quad \{Q_n, Q_b, Q_e\}}$$

$$T_{S_2} \equiv \frac{Tree_2}{\{Q\} \quad s_2 \quad \{R, R'_b, R'_e\}}$$

$$T_{try-finally} \equiv \frac{T_{S_1} \qquad T_{S_2}}{\{P\} \ \ \texttt{try} \ s_1 \ \texttt{finally} \ s_2 \ \ \{R'_n, R'_b, R'_e\}}$$

where

$$Q \equiv \left(\begin{array}{l} (Q_n \wedge \mathcal{X}Tmp = normal) \ \vee (Q_b \wedge \mathcal{X}Tmp = break) \vee \\ (\ Q_e[eTmp/excV] \wedge \mathcal{X}Tmp = exc \wedge eTmp = excV \ ) \end{array} \right)$$

$$R \equiv \left(\begin{array}{l} (R'_n \wedge \mathcal{X}Tmp = normal) \ \vee \ (R'_b \wedge \mathcal{X}Tmp = break) \vee \\ (R'_e \wedge \mathcal{X}Tmp = exc) \end{array} \right)$$

In this translation, the bytecode for s_1 is followed by the bytecode for s_2. In the translation of T_{S_1}, the `finally` block is added to the finally-list f with T_{S_2}'s source proof tree and its associated exception table. The corresponding exception table is retrieved using the function $getExcLines : Label \times Label \times ExcTable \rightarrow ExcTable$. Given two labels and an exception table et, $getExcLines$ returns, per every exception type in et, the first et's exception entry (if any) for which the interval made by the starting and ending labels includes the two given labels. Furthermore, a new exception entry, for the `finally` block, is added to the exception table et. Then, the bytecode proof for the case when s_1 throws an exception is created. The exception table of this translation is produced by the predecessor translations.

Let et', et'' be the following exception tables:

$$et_1 = et + [l_{start}, l_b, l_d, any]$$
$$et' = getExcLines(l_a, l_b, et_1)$$

Let b_{goto}, b_{store}, b_{load}, and b_{throw} be instructions specifications and B_{S_1}, B_{S_2}, and B'_{S_2} be bytecode proofs:

$$\begin{array}{rll}
[B_{S_1}, et_2] = & \nabla_S\ (T_{S_1},\ l_{start},\ l_b,\ l_{break},\ [T_{S_2}\ , et'] + f,\ et_1) & \\
[B_{S_2}, et_3] = & \nabla_S\ (T_{S_2},\ l_b,\ l_c,\ l_{break},\ f,\ et_2) & \\
b_{\mathsf{goto}} = & \{Q'_n\} & l_c : \mathsf{goto}\ l_{next} \\
b_{\mathsf{store}} = & \left\{ \begin{array}{l} shift(Q_e) \wedge \\ excV \neq null \\ \wedge\ s(0) = excV \end{array} \right\} & l_d : \mathsf{store}\ eTmp \\
[B_{S'_2}, et_4] = & \nabla_S\ (T_{S_2}, l_e, l_f, l_{break},\ f,\ et_3) & \\
b_{\mathsf{load}} = & \left\{\ Q'_n\ \vee\ Q'_b\ \vee\ Q'_e\ \right\} & l_f : \mathsf{load}\ eTmp \\
b_{\mathsf{throw}} = & \left\{ \begin{array}{l} (Q'_n\ \vee\ Q'_b\ \vee\ Q'_e) \\ \wedge\ s(0) = eTmp \end{array} \right\} & l_g : \mathsf{throw}
\end{array}$$

The translation is defined as follows:

$$\nabla_S\ (\ T_{try-finally},\ l_{start}, l_{next}, l_{break},\ f,\ et) = [\ B_{S_1} + B_{S_2} + b_{\mathsf{goto}} + b_{\mathsf{store}} + B_{S'_2} + b_{\mathsf{load}} + b_{\mathsf{throw}}\ ,\ et_4\ ]$$

It is easy to see that the instruction specifications b_{goto}, b_{store}, b_{load}, and b_{throw} are valid (by applying the definition of the weakest precondition). However, the argument for the translation of T_{S_1} and T_{S_2} is more complex. Basically, the result is a valid proof because the proof tree inserted in f for the translation of T_{S_1} is a valid proof and the postcondition of each finally block implies the precondition of the next one. Furthermore, for normal execution, the postcondition of B_{S_1} (Q_n) implies the precondition of B_{S_2} (Q).

5.3.4 Break Statement

To specify the rules for `break`, we use the following recursive function: *divide*: *ExcTable* × *ExcTableEntry* × *Label* × *Label* → *ExcTable*. Its definition assumes that the exception entry is in the given exception table and the two given labels are in the interval made by the exception entry's starting and ending labels. Given an exception entry y and two labels l_s and l_e, *divide* compares every exception entry, say x, of the given exception table to y. If the interval defined by x's starting and ending labels is included in the interval defined by y's starting and ending labels, then x must be divided to have the appropriate behavior of the exceptions. Thus, the first and the last interval of the three intervals defined by x's starting and ending labels, l_s, and l_e are returned, and the procedure is continued for the next exception entry. If x and y are equal, then recursion stops as *divide* reached the expected entry. The formal definition of *divide* is the following:

$$\begin{array}{l}
divide : ExcTable \times ExcTableEntry\ \times Label\ \times Label \rightarrow ExcTable \\
\quad divide : ([\ ], e', l_s, l_e) = [\ e'\ ] \\
\quad divide : (e : et, e', l_s, l_e) = \\
\qquad [\ l_{start},\ l_s,\ l_{targ},\ T_1\ ]\ + [\ l_e,\ l_{end},\ l_{targ},\ T_1\ ] + divide(et, e', l_s, l_e) \quad \textbf{if}\ e \subseteq e'\ \wedge\ e \neq e' \\
\qquad |\ e : et \quad \textbf{if}\ e = e' \\
\qquad |\ e : divide(et, e', l_s, l_e) \quad \textbf{otherwise}
\end{array}$$

where

$e \equiv [l_{start}, l_{end}, l_{targ}, T_1]$ and $e' \equiv [l'_{start}, l'_{end}, l'_{targ}, T_2]$

$$\begin{array}{l}
\subseteq\ : ExcTableEntry \times ExcTableEntry \rightarrow Boolean \\
\quad \subseteq\ : ([l_{start}, l_{end}, l_{targ}, T_1], [l'_{start}, l'_{end}, l'_{targ}, T_2]) = \\
\qquad |\ \ true \quad \textbf{if}\ (l'_{st} \leq l_{st})\ \wedge\ (l'_{end} \geq l_{end}) \\
\qquad |\ \ false \quad \textbf{otherwise}
\end{array}$$

When a `break` statement is encountered, the proof tree of every `finally` block the `break` has to execute upon exiting the loop is translated. Then, control is transferred to the end of the loop using the label l_{break}. Let $f_i = [T_{F_i}, et'_i]$ denote the i-th element of the list f, where

$$T_{F_i} = \frac{Tree_i}{\{U^i\} \;\; s_i \;\; \{V^i\}}$$

and U^i and V^i have the following form, which corresponds to the Hoare rule for `try-finally` (see Section 5.1):

$$U^i \equiv \left\{ \begin{array}{l} (U^i_n \;\wedge\; \mathcal{X}Tmp = normal) \vee \\ (U^i_b \;\wedge\; \mathcal{X}Tmp = break) \vee \\ \left(\begin{array}{l} U^i_e[eTmp/excV] \;\wedge\; \mathcal{X}Tmp = exc \wedge \\ eTmp = excV \end{array} \right) \end{array} \right\}$$

$$V^i \equiv \left\{ \left(\begin{array}{l} (V'^i_n \;\wedge\; \mathcal{X}Tmp = normal) \vee \\ (V'^i_b \;\wedge\; \mathcal{X}Tmp = break) \vee \\ (V'^i_e \;\wedge\; \mathcal{X}Tmp = exc) \end{array} \right), \;\; V^i_b, \;\; V^i_e \right\}$$

Let B_{Fi} be a *BytecodeProof* for T_{Fi} such that

$$[B_{F_i}, et_{i+1}] = \nabla_S \left(\begin{array}{l} T_{F_i}, \; l_{start+i}, l_{start+i+1}, l_{br}, f_{i+1}...f_k, \\ divide(et_i, et'_i[0], l_{start+i}, l_{start+i+1}) \end{array} \right)$$

$$b_{\mathsf{goto}} = \{B^k_b\} \quad l_{start+k+1} : \mathsf{goto}\ l_{br}$$

The definition of the translation is the following:

$$\nabla_S \left(\frac{}{\{P\} \;\; \texttt{break} \;\; \{false, P, false\}}, l_{start}, l_{next}, l_{br}, f, et_0 \right)$$

$$= [\; B_{F_1} + B_{F_2} + ...B_{F_k} + b_{\mathsf{goto}}, et_k]$$

To argue that the bytecode proof is valid, we have to show that the postcondition of B_{F_i} implies the precondition of $B_{F_{i+1}}$ and that the translation of every block is valid. This is the case because the source rule requires the break-postcondition of s_1 to imply the normal precondition of s_2.

The exception table has two important properties that hold during the translation. The first one (Lemma 1) states that the exception entries, whose starting labels appear after the last label generated by the translation, are kept unchanged. The second one (Lemma 2) expresses that the exception entry is not changed by the division. These properties are used to prove soundness of the translation.

Lemma 1 *If* $\nabla_S(\{P_n\}\ s\ \{Q\},\ l_a,\ l_{b+1},\ l_{break},\ f,\ et) = [(I_{l_a}...I_{l_b}), et']$ *and* $l_{start} \leq l_a < l_b \leq l_{end}$ *then for every* $l_s, l_e \in Label$ *such that* $l_b < l_s < l_e \leq l_{end}$ *and for every* $T \in Type$ *such that* $T \preceq Throwable \vee T \equiv any$*, the following holds:* $et[l_{start}, l_{end}, T] = et'[l_s, l_e, T]$.

Lemma 2 *Let* $r \in ExcTableEntry$ *and* $et' \in ExcTable$ *be such that* $r \in et'$*. If* $et \in ExcTable$ *and* $l_s, l_e \in Label$ *are such that* $et = divide(et', r, l_s, l_e)$*, then* $et[l_s, l_e, T] = r[2]$

5.4 Example

Figure 5.3 exemplifies the translation. The source proof of the example in Figure 5.2 is presented on the left-hand side and the corresponding bytecode proof on the right. An exception is thrown in the `try` block

with precondition $b = 1$. The `finally` block increases b and then executes a `break` changing the status of the program to break mode (the postcondition is $b = 2$). In the bytecode proof, the body of the loop is between lines 09 and 18. Lines 17 and 18 re-throw the exception produced at line 10. Due to the execution of a `break` instruction, the code from 17 to 18 is not reachable (this is the reason for their *false* precondition). The `break` translation yields at line 16 a goto instruction whose target is the end of the loop, *i.e.*, line 23.

```
void foo () {
    { true }
    int b = 1;
    { b = 1, false, false }
    while (true) {
      { b = 1, false, false }
      try {
          { b = 1, false, false }
          throw new Exception();
          { false, false, b = 1 }
      }
      finally {
          { b = 1 ∧Xtmp = exc }
          b = b+1;
          { b = 2 ∧Xtmp = exc, false,
            false }
          break;
          { false, b = 2 ∧Xtmp = exc,
            false }
      }
      { false, b = 2, false }
    }
    { b = 2, false, false }
    b = b+1;
    { b = 3, false, false }
}
```

Precondition	Instruction
{ **true** }	00 : **push** 1
$\{s(0) = 1\}$	01 : **store** b
$\{b = 1\}$	02 : **goto** 20
$\{b = 1\}$	09 : **new** *Exception*
$\{b = 1\}$	10 : **throw**
$\{b = 1 \wedge excV \neq null \wedge s(0) = excV\}$	11 : **store** *eTmp*
$\{b = 1 \wedge eTmp = excV\}$	12 : **push** 1
$\{b = 1 \wedge s(0) = 1\}$	13 : **load** b
$\{b = 1 \wedge s(1) = 1 \wedge s(0) = b\}$	14 : **binop** +
$\{b = 1 \wedge s(0) = b + 1\}$	15 : **store** b
$\{b = 2\}$	16 : **goto** 23
{ **false** }	17 : **load** *eTmp*
{ **false** }	18 : **throw**
$\{b = 1\}$	20 : **push true**
$\{b = 1 \wedge s(0) = \mathbf{true}\}$	21 : **iftrue** 04
$\{b = 2\}$	23 : **push** 1
$\{b = 2 \wedge s(0) = 1\}$	24 : **load** b
$\{b = 2 \wedge s(1) = 1 \wedge s(0) = b\}$	25 : **binop** +
$\{b = 2 \wedge s(0) = 1 + b\}$	26 : **store** b

Exception Table

From	*to*	*target*	*type*
0	7	10	*any*

Figure 5.3: Example of source and bytecode proofs generated by the Proof-Transforming Compiler.

5.5 Soundness Theorem

In a Proof-Carrying Code environment, a soundness proof is required only for the trusted components. PTCs are not part of the trusted code base: If the Proof-Transforming Compiler generates an invalid proof, the proof checker would reject it. But from the point of view of the code producer, we would like to have a compiler that always generates valid proofs. Otherwise, it would be useless.

We prove the soundness of the translations, *i.e.*, the translation produces valid bytecode proofs. It is, however, not enough to prove that the translation produces a valid proof, because the compiler could generate bytecode proofs where every precondition is false. The theorem states that if (1) we have a valid source proof for the statement s_1, and (2) we have a proof translation from the source proof that produces the instructions $I_{l_{start}} ... I_{l_{end}}$, their respective preconditions $E_{l_{start}} ... E_{l_{end}}$, and the exception table et, and (3) the exceptional postcondition in the source logic implies the precondition at the target label stored in the exception table for all types T such that $T \preceq \mathit{Throwable} \vee T \equiv \mathit{any}$ but considering the value stored in the stack of the bytecode, and (4) the normal postcondition in the source logic implies the next precondition of the last generated instruction (if the last generated instruction is the last instruction of the method, we use the normal postcondition in the source logic), (5) the break postcondition implies *finallyProperties*. Basically, the *finallyProperties* express that for every triple stored in f, the triple holds and the break postcondition

of the triple implies the break precondition of the next triple. And the exceptional postcondition implies the precondition at the target label stored in the exception table et_i but considering the value stored in the stack of the bytecode. Then, we have to prove that every bytecode specification holds ($\vdash \{E_l\}\ I_l$).

In the soundness theorem, we use the following abbreviation: for an exception table et, two labels l_a, l_b, and a type T, $et[l_a, l_b, T]$ returns the target label of the first et's exception entry whose starting and ending labels are less or equal and greater or equal than l_a and l_b, respectively, and whose type is a supertype of T.

Here, we present the theorem without the details of the properties satisfied by the finally function f. The proof runs by induction on the structure of the derivation tree for $\{P\}\ s_1\ \{Q_n, Q_b, Q_e\}$. The proof and the complete theorem can be found in our technical report [27].

Theorem 1

$$\left(\begin{array}{l} \vdash \dfrac{\mathit{Tree}}{\{P\}\quad s_1\quad \{Q_n, Q_b, Q_e\}} \equiv\ T_{S_1} \quad \wedge \\ [(I_{l_{start}} ... I_{l_{end}}), et] = \nabla_S\,(T_{S_1},\ l_{start}, l_{end+1}, l_{break}, f, et') \wedge \\ (\forall\ T : \mathit{Type} : (T \preceq \mathit{Throwable} \vee T \equiv \mathit{any}) : (Q_e \wedge \mathit{excV} \neq \mathit{null} \wedge\ s(0) = \mathit{excV})\ \Rightarrow\ E_{et'[l_{start}, l_{end}, T]}) \wedge \\ \big(Q_n \quad \Rightarrow \quad E_{l_{end+1}}\big) \quad \wedge \\ (Q_b\ \Rightarrow \mathit{finallyProperties}) \end{array}\right)$$

$$\Rightarrow$$

$$\forall\ l\ \in\ l_{start}\ ...\ l_{end} : \vdash \{E_l\}\ I_l$$

Chapter 6

Implementation issues

6.1 Architecture refinement

6.1.1 Mobius PVE

The MOBIUS PVE is the tool that will host the proof-transforming compiler components. It is based upon Eclipse [15], a development environment for Java. One of the recent additions made to this environment was the ability to brand applications out of it, creating specific rich client platforms. The source VCGen will be based upon some already existing components of MOBIUS PVE:

- ESC/Java2 and, in particular, its source to AST compiler, JavaFE, and its multi-prover back-end [16, 10, 17]
- Bico, a Java to Bicolano translator [24]
- Eclipse and ProverEditor, the multi-prover editor of the MOBIUS PVE, which are both end-user components [36]

ESC/Java2 ESC/Java2 is a mature extended static checker for Java and JML. Its new version will be the core component of the MOBIUS PVE. Two parts of it are interesting for the direct VCGen to work, first the parser, and second the multi-prover output.

Right now, the parser is a specific Java and JML parser built on top of the JavaFE parser, which is one of the ESC/Java2 components that should be changed in the final version of the MOBIUS PVE. This component generates an AST that can be visited using a visitor pattern. The visitor pattern that should be used is generic enough to be applied to another AST once the parser will be changed in a near future.

ESC/Java2 provides a new AST to express first order logic formulas. The formulas are typed, which is important for the generation of Coq verification conditions.

Bico Bico is a translator from Java bytecode to Bicolano's Coq version of Java bytecode. It is used to translate the bytecode to Bicolano instructions. For the direct bytecode VCGen, Bico generates towards a map implementation of Bicolano's axiomatisation.

Bico generates mainly two groups of files: the class translation files and the summary files, which will be used by the direct VCGen. The class translation files have names prefixed by the name of the class and contain the Bicolano class name, as well as the field and method signatures, and the interesting part for the methods: the methods' body. The summary files gather all the types (the class and interface name), the method and field signatures and the body, into an accessible list that should be used further on.

Environment The direct VCGen is included as a component in the MOBIUS PVE. It is included inside of Eclipse, which enables it to use some Eclipse facilities: it is project targeted and it generates the file to a designated project directory. There will be soon a GUI component to easily view the verification conditions

status (it should be a generic one compatible with ESC/Java2). The classpath is handled as well by Eclipse, and it is done in a uniform way for the different tools (ESC/Java2, Bico and the direct VCGen).

More specifically, the MOBIUS PVE already contains some components to edit Coq files with the MOBIUS PVE component ProverEditor. This component is a lightweight multi-prover editor for Eclipse. It was previously used in Jack [19, 5] to handle the user-interaction. This component will be useful especially for case studies.

6.1.2 Concrete architecture of the direct VCGen

The core component which is developed for the proof-transforming compiled is the direct VCGen. It is called direct because, as opposed to other approaches used in MOBIUS PVE, it does not use any intermediate language to compute the weakest precondition of a program. The direct VCGen's implementation is arranged into two main parts, the verification condition generation for bytecode, which has been proven correct against Bicolano semantic, and the VCGen for source code. The whole direct VCGen implementation takes JML annotated source code as input and generates two verification conditions, one for source and the other for bytecode. As described in Chapter 4, we arrange for the proofs of these VCs to be nearly identical.

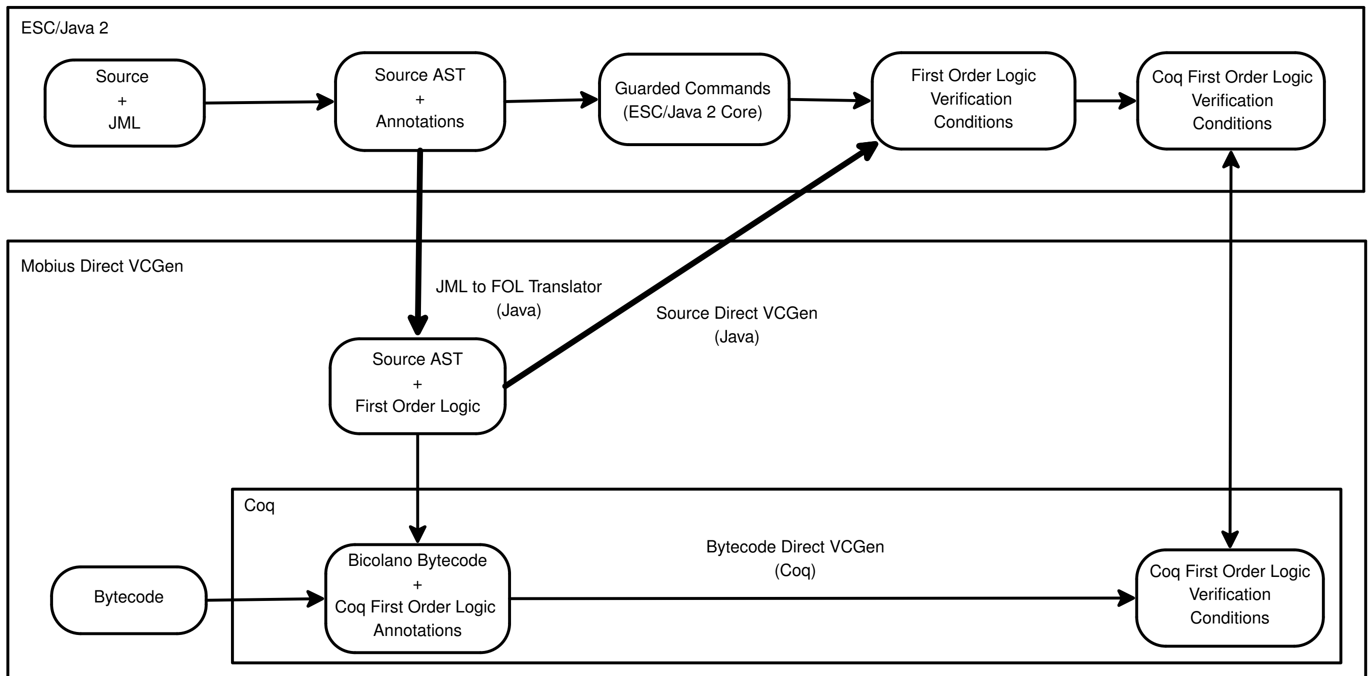


Figure 6.1: The direct VCGen inclusion inside MOBIUS PVE architecture. Boxes denote data, arrows denote computations. The fat arrows represent the implementation work done in this task so far.

The source subsystem

The direct VCGen for source program is plugged directly into ESC/Java2. As entry, it takes JML annotated source code, this source code is then parsed by ESC/Java2 into an AST with some annotation nodes. Then the annotations are translated into first order logic, using the JML to FOL translator described in Chapter 3. This translator produces an FOL annotated AST. Then the verification conditions are generated using the direct VCGen for source programs. The verification conditions are simple first order logic formulas in the same format as the one ESC/Java2 uses. These formulas are generated in Coq format. The whole source processing is implemented in Java.

The bytecode subsystem

This subsystem is the one that is formalized in Coq. The verification conditions for bytecode are generated from a direct VCGen written in the proof assistant's language. The first step is to translate the bytecode (which has been compiled from the source code using `javac` or a non-optimizing compiler) and the annotations to Coq. The bytecode is translated using bico, the annotations are translated from the resulting first order logic formula gotten from the JML to FOL translator. Once the bytecode and the annotations are translated to Bicolano, the direct bytecode VCGen [23, 24] generates from this input some verification conditions, which are very similar to those of the source code.

The direct VCGen for bytecode is based on the simple verification condition generator, which is presented in Chapter 4. The verification conditions are generated directly as lemmas to prove on the Coq level. The VCGen is deeply embedded in Coq, and it has some optimizations.

One of the difficulties for the inclusion was that no standard way of testing was set. The tool could only be tried on examples built by hand. Hopefully, this has been solved by adapting Bico to generate Bicolano bytecode together with a translation to Coq of the first order logic annotations presented in this report. The extension with the annotation has been done given the hypothesis that the Java compiler was non-optimizing. Still, this part is just an experimental extension for testing purposes. The annotations are generated as functions, with their parameters being the local variables. It is done in a slightly more generic way as what was presented as Coq annotations in Deliverable 3.1 [23]. This extension enabled the testing of the VCGen on more various examples, and more specifically a better synergy with the source direct VCGen.

6.2 Source Logic for the PTC

The direct VCGen for source programs treats all the main Java source constructions, except multi-dimensional arrays and concurrency. It is based on a simple calculus, as described in Chapter 4, which discharges the verification conditions as side conditions when it encounters a loop construct or an assert construct. It takes as input the Java source with JML annotations.

6.2.1 Pre-processing

In the pre-processing phase, the Java and JML program are parsed, an AST is built using JavaFE, and the JML annotations are transformed to first order logic as described in Chapter 3.

JavaFE and ESC/Java2 parsing

We have seen in Section 6.1.1 that JavaFE provides an AST which is pretty standard. It enables a visitor design pattern and some standard decorations. The main class file which is subtyped by all the nodes is `javafe.ast.ASTNode`.

There are two ways of using visitors with this AST. Returning nothing (`void accept(Visitor)`) or returning a result (`Object accept(VisitorArgResult, Object)`). A difference is also made between the pure Java AST (defined only in JavaFE) and the JML annotated AST which has its base visitor defined in the package `escjava.ast`. If the visitor used is not the one from ESC/Java2, it would simply ignore the JML constructs.

The decoration of the AST is done with a decorator. The only decoration which is used at this stage is the type decoration. Each node of the AST contains an array that is filled with decorations. To add more information to the AST, one only has to use or subclass `javafe.ast.ASTDecoration`.

Once the AST has been created and type-checked, it can be processed through Mobius' direct VCGen.

First order logic

We use the first order logic that is used by ESC/Java2 to generate proof obligations. It is defined in the package `escjava.sortedProver`. This logic is multi-sorted and provides eight different types (`SPred`, `SAny`,

SMap, SValue, SBool, SInt, SReal, SRef). There is no way to define real custom types over these basic types as they are hard-coded. Another problem arises for custom predicates: there is no standard programmatic way to build custom predicates or functions. Therefore, we had to modify the code of this backend to use the MOBIUS memory model.

Another drawback of the ESC/Java2 backend was the fact that the typed intermediate representation (before the translation to prover-specific formulas) was programmed using non-static inner classes. This makes the overall backend less usable for other purposes, especially to extend it with substitution for all the provers at the same time. There was now way of creating this inner datastructure from the outside of the backend. This problem was partially solved by writing a library layer to handle all these creation. The library layer consists of eight classes, corresponding to the eight types of the logic. Each class is parameterized by an instance of the class which contains the inner-classes representation.

Despite these small issues, it was important to use the ESC/Java2 backend because it permits the direct VCGen to be properly integrated in MOBIUS PVE as well as to gain multi-prover output.

JML to first order logic

After parsing and type checking, we obtain the JML and Java typed AST. At this point, the first order logic formulas have to be computed from of the JML annotations. The JML nodes will be removed and the AST nodes will be decorated using the class `AnnotationDecoration`. An annotation decoration contains two elements: the list of annotations, which are evaluated before the node, and the list of annotations, which have to be evaluated after the node. The annotations that are part of the list are of three possible kinds: **assert**, **assume**, or **set**. Loop invariants are added directly to the node if needed.

The translation from JML to first order logic is done in the package `formula.jmlTranslator`. It is based upon several visitors, which perform the translation described in Chapter 3. Once the translation is done, no more JML nodes appear in the AST as they are replaced by first order logic terms.

Method specifications are handled differently. They are all concentrated in a single location, the `Lookup` class. This class is accessed through static calls and the translation of pre- or post- condition of the methods is done there.

6.2.2 Verification conditions generation

Once the specifications are translated and the AST has been decorated, the verification conditions can be computed. This is done using several visitors. The weakest precondition calculus from Chapter 4 is implemented in the package `vcgen`.

Basic principle

A typical scenario for the weakest precondition of an instruction is the following:

- first, the annotations that decorate the after-part of the instruction are taken into account,
- then the weakest precondition of the instruction is computed, and
- finally, the annotations that decorate the before-part of the instruction are taken into account.

The data structure used to pass the postconditions is complex because of the number of possible postconditions. It must contain the normal and exceptional postconditions, which are the ones from the current method. There are several postconditions that change depending on the content of the method. First, there are the try catch block exceptional postconditions. The list of these exceptional postconditions is indexed by the type of the exception which are caught. There are also several postconditions which must be stored as well, like the `continue`, `continue label` postconditions and the `break` and `break label` postconditions.

$$\frac{}{\mathsf{wp}_{\mathcal{S}}(\mathsf{assert}\ \phi,\ \psi) = \phi \wedge \psi, \emptyset} \qquad \frac{}{\mathsf{wp}_{\mathcal{S}}(\mathsf{assume}\ \phi,\ \psi) = \phi \rightarrow \psi, \emptyset}$$

$$\frac{}{\mathsf{wp}_{\mathcal{S}}(\mathsf{set}\ v,\ \psi) = \forall v.\psi, \emptyset} \qquad \frac{}{\mathsf{wp}_{\mathcal{S}}(\mathsf{set}\ v\ val,\ \psi) = \psi\{v \mapsto val\}, \emptyset}$$

Figure 6.2: Weakest precondition of the first order logic constructs generated for local JML annotations

$$\frac{\mathsf{lookup}(\psi, \mathsf{break}) = \psi_{brk}}{\mathsf{wp}_{\mathcal{E}}(\mathsf{break},\ \psi) = \psi_{brk}, \emptyset} \qquad \frac{\mathsf{lookup}(\psi, lbl) = \psi_{lbl}}{\mathsf{wp}_{\mathcal{E}}(\mathsf{break}\ lbl,\ \psi) = \psi_{lbl}, \emptyset}$$

Figure 6.3: Weakest precondition for the break instruction

Specific annotations

The annotations generated by the translation from local JML specifications to FOL have a specific weakest precondition calculus. The calculus presented in Figure 4.1 is easily extended to treat the local annotations used (Figure 6.2).

The weakest precondition for `assert` and `assume` is standard. The `set` construct is two-fold. There is the declaration `set`, which takes only one argument, and the assignment `set`, which takes two arguments. The value val is computed during the JML to FOL translation.

Expressions

The weakest precondition calculus for the expressions is based on the calculus for JAVA$_{\mathcal{E}}$(Chapter 4). The implementation is split into three classes: the visitor, which inspects each node of the expression, two delegate objects, one generic that computes the weakest precondition for binary expressions, and one that computes it for all other expressions such as increment instructions, initialization expressions, or method invocations. The implementation is standard as the way exceptions are treated is using the usual handlers with `try-catch` constructs that are described in JAVA$_{\mathcal{E}}$.

Handling break and continue

Breaks and continues are difficult to handle, because they add complexity to the postcondition structure. As shown in Chapter 5, these constructs require specific lists of postconditions.

Each time a loop, a `try-catch`, or a block is entered by the weakest precondition calculus, the slot in the postcondition data structure for the `break` is filled with the current normal postcondition. This postcondition will be the postcondition for the `break` if encountered in the block. If there is a label to the block, the postcondition is put in a list of postcondition indexed by labels. This is shown in Figure 6.3, where the function lookup is used to retrieve the `break` postconditions in the postcondition data structure. The same treatment is given for the `continue` postconditions.

6.2.3 A backend to Bicolano

A crucial concern for the correspondence between source and bytecode verification conditions was to use the Bicolano memory model for both VCGen. In order to output Coq compatible proof obligations, a plugin to pretty print towards Coq was written for the ESC/Java2 backend.

Sorted prover's Coq backend

The main differences between Bicolano and the sorted prover generic backend lies in the handling of type names. In Bicolano, type names have a specific sort, and in order to be used as types in a generic manner, they have to be converted. For instance, the type `Object` in Bicolano would be used in a proof obligation as `(ReferenceType (ClassType Object.className))`. In the sorted prover, the only way to express the type is using a variable of type `Ref`, here it would be `T_java_lang_Object`. When used with the same construct, `typeof`, this difference naturally leads to typing errors.

An issue also appears with the handling of heap access. It is also due to the handling of types. The predicates in Bicolano and the sorted prover to access the heap are different because in Bicolano, the heap access methods take an addressing mode data structure, whereas in the backend, it is once again a reference variable. Here the problem is for field names, but the result is the same: the typing of the construction cannot be properly verified if the backend generically, like its intended purpose.

The solution to these problems we apply is a temporary one: it is to simply separate the plugin which pretty prints toward Coq from the rest of ESC/Java2. With this solution, the direct VCGen can work properly, but we lose some features that the integration in ESC/Java2 was offering.

Prelude generation

The usual prelude generation for ESC/Java2 is simple, as ESC/Java2 generates all the background predicates at runtime, when it generates the proof obligations. Here, this was not possible since we want to be compatible with Bicolano. All the constructs that are used are already declared in Bicolano's axiomatization. The only elements that are not properly present are the class and field names. For this purpose Bico has been used. It properly generates all the names needed for proof obligations, as well as instantiate properly the axioms with an implementation using the `Map` data structure. Bico gives a generic way to generate the prelude for Coq verification conditions.

Chapter 7

Type-preserving compilation for a type system for secure information flow

JFlow [29] offers a practical tool for developing secure applications, and in particular for ensuring to developers that applications meet high-level policies about API usage. In contrast, the type system for bytecode developed in Task 2.1. and [6] augments the Java security architecture to provide assurance to users that applets respect high-level policies about API usage.

In this chapter we connect them via a type preservation result, showing that programs typable in a suitable fragment of JFlow will be compiled into bytecode programs that pass information flow bytecode verification (as suggested, e.g., by Abadi [1]). The interest of such a result is to show on the one hand that applications written with JFlow can be deployed in a mobile code architecture that delivers the promises of JFlow in terms of confidentiality, and on the other hand that the enhanced security architecture can benefit from practical tools for developing applications that meets the policy it enforces.

7.1 Control dependence regions

Tracking information flow via control flow in a structured language without exceptions is easy since the analysis can exploit control structure [?, 2]. Exceptions make tracking cumbersome due to the loss of structured control flow. Our high level type system tracks exception levels in a way similar to [9].

For unstructured low level code implicit flows can be tracked in terms of an analysis of *control dependence regions* which gives information about different blocks in the program due to conditional or exceptional instructions. This analysis can be statically approximated [7, 38].

To deal with this mismatch between tracking of implicit flows at source and bytecode level, we introduce an intermediate analysis that applies to source code but uses control dependence regions. One of the contributions of this chapter is an inductive definition of control dependence regions for a language with exceptions. An important consequence of such an analysis for source programs is that, given a compiler from source programs to target programs, it is possible to obtain control dependence regions for the compiled program (Definition 7.7.1). In a scenario where the compiler is not trusted, it is possible to check that the regions given by the analysis based on compilation has the requisite properties for soundness of the information flow analysis (see SOAP Property from Task 2.1 and [6]).

7.2 High level security type system

In this section we define a high level type system for the source language. By high level we mean that it consists of syntax-directed rules together with subsumption rules. It enforces the policy specified by security method signatures and a mapping, ft, from field names to security levels.

Method signatures are of the form

$$\vec{k_v} \xrightarrow{k_h} \vec{k_r}$$

where $\vec{k_v}$ provides the security level of the method arguments (and to all intermediate variables used in the method), k_h is the effect of the method on the heap, $\vec{k_r}$ (called *output level*) is a list of security levels of the form $\{n : k_n, e_1 : k_{e_1}, \ldots e_n : k_{e_n}\}$, where k_n is the security level of the return value and e_i is an exception class that might be propagated by the method in a security environment (or due to an exception-throwing instruction) of level k_i. In the rest of the chapter we will write $\vec{k_r}[n]$ instead of k_n and $\vec{k_r}[e_i]$ instead of k_{e_i}.

Source language. Basically, we operate on a subset of JAVA$_\mathcal{E}$. In the following, we shortly summarize the constructs that are used in this chapter. Exceptions in Java can be explicitly programmed and are also thrown by expressions such as field access (null dereference) and type cast (cast failure). We use a desugared source language in which exceptions can only occur in specific syntactic forms. This helps simplify the formalization in Section 7.3 where we attach control flow labels to commands that can branch due to exceptions. We also require method bodies to end with a return, to simplify the definition of control flows. Any command can be desugared to this form using additional local variables.

The grammar is as follows; x represents any variable name, v a literal value, C a class name, f a field name, m a method identifier and op represents an arithmetic or boolean operation.

$$\begin{array}{rcl} e & ::= & x \mid v \mid e\ op\ e \mid \mathsf{new}\ C \\ c & ::= & x = e \mid x = e.f \mid e.f = e \mid \mathsf{throw}\ e \mid \\ & & \mathsf{return}\ e \mid c; c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \\ & & \mathsf{while}\ e\ \mathsf{do}\ c \mid \mathsf{try}\ c\ \mathsf{catch}(X\ x)\ c \mid x = e.m(\vec{e}) \\ p & ::= & [\mathsf{m}\ (\vec{x})\{c; \mathsf{return}\ e\}] \end{array}$$

A program p is a list of method declarations of the form $\mathsf{m}\ (\vec{x})\{c; \mathsf{return}\ e\}$, where c is a command that does not contain return instructions, and $\vec{x}$ is the set of local variables used in the body of the method.

We assume that every program is closed under the successor relation.

We use the term *exception-throwing* for commands of the form $e.f = e'$ or $x = e.f$ or $x = e.m(\vec{e})$ or throw e. Handlers for exceptions appear in the catch part of a try-catch command.

Source labels and control flows. To name *program points* where control flow can branch or writes can occur, we add natural number labels to the source syntax —not to be confused with security labels given by the policy.

$$\begin{array}{rcl} c & ::= & [x = e]^n \mid [x = e.f]^n \mid [e.f = e]^n \mid [\mathsf{throw}\ e]^n \mid \\ & & [\mathsf{return}\ e]^n \mid c; c \mid [\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c]^n \mid \\ & & [\mathsf{while}\ e\ \mathsf{do}\ c]^n \mid [\mathsf{try}\ c\ \mathsf{catch}(X\ x)\ c]^n \mid [x = e.m(\vec{e})]^n \end{array}$$

The notation for labels of compound commands is convenient but visually misleading in that the label pertains to the branching point in the control flow graph, for if and while, and the start of the handler for try-catch.

By contrast with Nielson et al. [18], we do not need to label expressions. The significance of labeling commands will become clear later in the chapter, when we give a definition for control dependence regions for programs as mapping from branching commands (such as if, while or exception-throwing commands) to sets of labels, corresponding to commands included in their regions. Throughout this section, we only use labels for the command throw (to define a class analysis). We write a command together with its label, $[c]^n$, only when the label is relevant for the context.

Remark on notation. We write $\leq$ for the lattice ordering on the set $\mathcal{S}$ of security levels and $\sqcup$ for least upper bounds. The latter is extended to $\emptyset$ by defining $\emptyset \sqcup k = k$.

The join operation $\sqcup_E$ for two exception effect lists is the join operation for security levels discriminated by class of exception, e.g. $\{e_1 : k_{e_1}, e_2 : k_{e_2}\} \sqcup_E \{e_2 : k_{e_2'}, e_4 : k_{e_4}\} = \{e_1 : k_{e_1}, e_2 : k_{e_2} \sqcup k_{e_2'}, e_4 : k_{e_4}\}$.

$$\frac{}{\vec{k_v} \vdash x : \vec{k_v}(x)}\ \text{VAR} \qquad \frac{}{\vec{k_v} \vdash v : L}\ \text{VAL} \qquad \frac{}{\vec{k_v} \vdash new\ C : L}\ \text{NEW}$$

$$\frac{\vec{k_v} \vdash e : k \qquad \vec{k_v} \vdash e' : k}{\vec{k_v} \vdash e\ op\ e' : k \sqcup k'}\ \text{OP} \qquad \frac{\vec{k_v} \vdash e : k \quad k \leq k'}{\vec{k_v} \vdash e : k'}\ \text{SUBSUME}$$

Figure 7.1: High level typing rules for source language expressions.

Type system. In presence of exceptions, commands can have multiple exits and the region of a branch need not be contained within the command. We define *high level judgments* $\vdash c : k_1, \vec{k_2}$ with the meaning that c is secure and writes variables/fields of level at least k_1. Moreover, the information revealed by the termination mode is given by $\vec{k_2}$, where $\vec{k_2}$ is a list of security levels of the form $\{\mathbf{e_1} : k_{e_1}, \ldots, \mathbf{e_n} : k_{e_n}\}$ where k_{e_i} is a security level and $\mathbf{e_i}$ is an exception class. Furthermore, the judgment allows $\vec{k_2}$ to be $\emptyset$ which indicates that no exception can escape from the command. In the case of exception-throwing commands, the exception effect is given by the type of the expression which might cause the exception to be thrown, e.g., in the case $x = e.f$ the exception effect k is the type of expression e (see rule [ASSIGN2]).

Figure 7.1 gives the typing rules for source language expressions . These are used both in the high level typing system and in the intermediate system. Figure 7.2 gives the high level typing rules for source programs and Figure 7.3 shows the rule of typability for method body.

Notice that commands such as $x = e$ or return e are typed with exception effect $\emptyset$, meaning that these commands cannot throw exceptions.

The type system is parameterized by a class analysis and an exception analysis, as described in Task 2.1.

Exception effects are used to impose constraints on successor commands, e.g., in $c; c'$ exception effects of c must be less or equal than the write effect of c' (see $k_1 \leq k'$). In general, exception effects of a command restrict the write effect of its successor commands, except in the case of try-catch. In the [CATCH] rule, the exception effect for the command in the try part imposes a constraint on the type of variable x (that stores the exception object) and imposes a constraint on the write effect of the code of the handler (catch part). If the catch part cannot throw new exceptions, i.e. its exception effect is $\emptyset$, and all exceptions in the try part are of the class of the handler (i.e. X in the rules), then there are no constraints on the write effects of successor commands of try-catch.

The following example illustrates how exception-throwing commands inside while commands can lead to information leaks.

Example 1 (exceptions in while) *Let x and x' be high variables and y be low. The program*

$$\text{while } x \leq 3 \text{ do } \{x' = y.f; x = x + 1;\} \text{ return } e$$

is interferent because $x' = y.f$ can throw an exception and there is no handler for it. Suppose that the variable y is initially null. Then the program terminates in an abnormal state if the (high) expression $x \leq 3$ is true and it terminates in a normal state if $x \leq 3$ is false. This program will be rejected by the [WHILE] rule, which does not allow low exceptions in high environments (constraint $k = k_e$). Now assume that variable x is low and x' and y are high. Again the program is interferent: A high null pointer exception can be thrown so the program will terminate abnormally depending on the value of high variable y.

This program will also be rejected by the [WHILE] rule.

The following lemma claims that any subcommand of a command c has at least the same write effect as that of c.

Assign
$$\frac{\vec{k_v} \vdash e : k \qquad k \leq \vec{k_v}(x)}{\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash x = e : \vec{k_v}(x), \emptyset}$$

Seq
$$\frac{\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c : k, \vec{k_1} \qquad \Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c' : k', \vec{k_1'} \qquad \forall e : k_1 \in \sqcup_J(k_1)\ k_1 \leq k'}{\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c\, ;\, c' : k \sqcap k', \vec{k_1} \sqcup_E \vec{k_1'}}$$

While
$$\frac{\vec{k_v} \vdash e : k \qquad \Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c : k, \vec{k'} \qquad \vec{k'} \neq \emptyset \Rightarrow k = k_e \qquad k_e = \sqcup\{k_1 | \mathbf{e} : k_1 \in \vec{k'}\}}{\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash \mathsf{while}\ e\ \mathsf{do}\ c : k, k_e}$$

Cond
$$\frac{\vec{k_v} \vdash e : k \qquad \Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c : k, \vec{k'} \qquad \Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c' : k, \vec{k'} \qquad k_e = \sqcup\{k_1 | \mathbf{e} : k_1 \in \vec{k'}\} \qquad \vec{k'} \neq \emptyset \Rightarrow k \leq k_e}{\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c' : k, k_e}$$

Assign2
$$\frac{\vec{k_v} \vdash e : k \qquad k \sqcup \mathsf{ft}(f) \leq \vec{k_v}(x)}{\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash x = e.f : \vec{k_v}(x), \mathbf{np} : k}$$

Update
$$\frac{\vec{k_v} \vdash e : k \qquad \vec{k_v} \vdash e' : k' \qquad k \sqcup k' \leq \mathsf{ft}(f)}{\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash e.f = e' : \mathsf{ft}(f), \mathbf{np} : k}$$

Catch
$$\frac{\begin{array}{c}\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c : k, \vec{k_1} \qquad (\forall X' : k_{x'} \in \vec{k_1}. X' \leq X. k_{x'} \leq k \sqcap \vec{k_v}(x)) \\ \Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c' : k, \vec{k_1'} \qquad (\forall X' : k_{x'} \in \vec{k_1}. X' \not\leq X. k_{x'} \leq \vec{k_u}\ )\end{array}}{\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash \mathsf{try}\ c\ \mathsf{catch}\ (X\ x)\ c' : k \sqcap \vec{k_v}(x), \vec{k_1'} \sqcup_E \vec{k_u}}$$

Throw
$$\frac{\vec{k_v} \vdash e : k \qquad \vec{X} = \mathsf{classAnalysis}(n)}{\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash [\mathsf{throw}\ e]^n : H, \vec{X} : k}$$

Invoke
$$\frac{\begin{array}{c}\forall i \in [0, \mathsf{length}(\vec{e}) - 1].\ \vec{k_v} \vdash \vec{e}[i] : \vec{k_v}[i] \qquad \mathsf{mt}_{m'} = \vec{k_v'} \xrightarrow{k_h'} \vec{k_r'} \\ \vec{k_v'}[0] \sqcup k_h \leq k_h' \qquad \vec{k_r'}[n] \leq \vec{k_v}(x) \qquad k_e = \{\mathbf{e} : k | \mathbf{e} : k \in \vec{k_r'}\ \wedge\ \mathbf{e} \neq n\}\end{array}}{\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash x = e.m'(\vec{e}) : \vec{k_v}(x) \sqcap k_h', \vec{k_e}}$$

Return
$$\frac{\vec{k_v} \vdash e : k \qquad k \leq \vec{k_r}[r] \quad k_e = \sqcup\{k_1 | \mathbf{e} : k_1 \in \vec{k_r}\ \wedge\ \mathbf{e} \neq n\}}{\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash \mathsf{return}\ e : k_e, \emptyset}$$

Subsume2
$$\frac{\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c : k_1, \vec{k_2} \qquad k_1' \leq k_1 \qquad \vec{k_2} \leq \vec{k_2'}}{\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c : k_1', \vec{k_2'}}$$

Figure 7.2: High level typing rules for source language command.

Typability of method body

$$\frac{\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c : k, \vec{k'} \quad \forall X : k_x \in \vec{k'}. : k_x = \vec{k_r}[\mathsf{e}] \quad k_e = \sqcup\{k_1 | \mathbf{e} : k_1 \in \vec{k'}\} \quad k_e \leq \vec{k_r}[\mathsf{r}] \quad k_h \leq k}{\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c; \mathsf{return}\ e}$$

Figure 7.3: Typability for method body.

Lemma 19 *Suppose* $\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c : k_1, \vec{k_2}$ *and* $\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c' : k_1', \vec{k_2'}$. *Let* c' *be a subcommand of* c. *Then* $k_1 \leq k_1'$ *and if* c' *throws an exception of type* $\mathbf{e}$ *and has no handler inside* c *then* $\vec{k_2'}[\mathsf{e}] \leq \vec{k_2}[\mathsf{e}]$.

Example 2 (Example with try-catch) *Consider the command*

$$\mathsf{try}\ [\mathsf{throw}\ x]^n;\ x = 1;\ \mathsf{catch}\ (X\ y)\ y = null$$

where variable x *is high* ($\vec{k_v}(x) = H$) *and variable* y *is low*($\vec{k_v}(x) = L$). *Suppose that a sound class analysis for* n *returns* $\{X\}$. *The program then is interferent; indeed its effect is the same as the direct assignment* $y = x$. *It is rejected by the type system because of the [*CATCH*] rule, where the exception effect of the try part (here,* H*) has to be less or equal than the type of the variable in catch (here,* L*).*

Typability. A method m is typable w.r.t. a method signature table Γ, a global field policy ft, and a signature *sgn* if its body is typable with rule in Figure 7.3. A program is typable, when all its methods are typable.

Example 3 (Example of high unhandled exceptions) *Let* x *be a high variable* ($\vec{k_v}(x) = H$) *and* y *be a low variable*($\vec{k_v}(x) = L$). *The method code* $x = y.f;\mathsf{return}\ e$ *with signature that includes an exception list* $\{\mathbf{np} : L\}$ *is interferent because a high can be thrown and there is no handler for it and the program will terminate normally or abnormally depending on high variable* y. *In our type system, this program is rejected because typability rule requires that the join of exception effect levels of the first command be less or equal then the write effect of the* return *command, that for this case is* L.

7.3 Intermediate type system for source code

In this section we introduce another type system for the source language. The *intermediate type system* serves as a bridge between the high level type system and the type system for the target language.

The function labels takes a command and returns all the labels of its subcommands, e.g. $\mathsf{labels}([x = e]^n) = \{n\}$, and $\mathsf{labels}(c; c') = \mathsf{labels}(c) \cup \mathsf{labels}(c')$.

Labels on program points are used in *intermediate judgments* of the form $\vdash c : E$. Here E is a *security environment*, i.e., a function $E : \mathsf{labels}(SP_m) \to \mathcal{S}$ that assigns levels to all program points of a method body SP_m. The body of a method m, SP_m, is typable with respect to a signature sgn and a function sregion, written $\Gamma, \mathsf{sregion}, \mathsf{sgn} \vdash SP_m : E$, if its command part is typable with respect to E according to the rules given in Figure 7.5. Roughly, the idea is that for a field or variable assignment with control label n, the field or variable must have level at least $E(n)$. Note that the rules recurse on the structure of constituent commands c of SP_m, but a single E is used throughout. A program is typable if all its methods are typable with all of its signatures.

The constraints in the rules involve control dependence regions (sregion), which we now proceed in several steps to define.

We use the notation $C[-]$ to denote context of a command. We use square brackets both for labeling and for contexts; it should be clear that $[c]^n$ without a capital letter in front means that command c has label n, whereas $C[c]$ with a capital C in front means $C[-]$ is the context of command c.

c	$init(c)$	$final(c)$	$except(c)$
$[x = e]^n$	n	$\{n\}$	$\emptyset$
$[x = e.f]^n$	n	$\{n\}$	$\{(\mathbf{np}, n)\}$
$[e.f = e]^n$	n	$\{n\}$	$\{(\mathbf{np}, n)\}$
$[\mathsf{throw}\ e]^n$	n	$\emptyset$	$\{(\mathbf{e}, n) \mid \mathbf{e} \in \mathsf{classAnalysis}(n)\}$
$c_1; c_2$	$init(c_1)$	$final(c_2)$	$except(c_1) \cup except(c_2)$
$[\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]^n$	n	$final(c_1) \cup final(c_2)$	$except(c_1) \cup except(c_2)$
$[\mathsf{while}\ e\ \mathsf{do}\ c_1]^n$	n	$\{n\}$	$except(c_1)$
$[\mathsf{try}\ c_1\ \mathsf{catch}(X\ x)\ c_2]^n$	$init(c_1)$	$final(c_1) \cup final(c_2)$	$except(c_2) \cup \{\mathbf{e}, n) \mid \mathbf{e} \in (except(c_1) \wedge \mathbf{e} \not\leq X)\}$
$[\mathsf{return}\ e]^n$	n	$\{n\}$	$\emptyset$
$[x = e.m(\vec{e})]^n$	n	$\{n\}$	$\{(\mathbf{e}, n) \mid \mathbf{e} \in \mathsf{excAnalysis}(m)\}$

c	$flow(c)$
$[x = e]^n$	$\emptyset$
$[x = e.f]^n$	$\emptyset$
$[e.f = e]^n$	$\emptyset$
$[\mathsf{throw}\ e]^n$	$\emptyset$
$c_1; c_2$	$flow(c_1) \cup flow(c_2) \cup \{(n, init(c_2)) \mid n \in final(c_1)\}$
$[\mathsf{try}\ c_1\ \mathsf{catch}(X\ x)\ c_2]^n$	$flow(c_1) \cup flow(c_2) \cup \{(n', n) \mid (\mathbf{e}, n') \in except(c_1) \wedge \mathbf{e} \leq X\} \cup \{(n, init(c_2))\}$
$[\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]^n$	$flow(c_1) \cup flow(c_2) \cup \{(n, init(c_1)), (n, init(c_2))\}$
$[\mathsf{while}\ e\ \mathsf{do}\ c_1]^n$	$flow(c_1) \cup \{(n, init(c_1))\} \cup \{(p, n) \mid p \in final(c_1)\}$
$[\mathsf{return}\ e]^n$	$\emptyset$
$[x = e.m(\vec{e})]^n$	$\emptyset$

Figure 7.4: Forward flows.

Assign

$$\frac{\vec{k_v} \vdash e : k \qquad k \sqcup E(n) \leq \vec{k_v}(x)}{\Gamma, \mathsf{region}, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash [x = e]^n : E}$$

Seq

$$\frac{\Gamma, \mathsf{region}, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c : E \qquad \Gamma, \mathsf{region}, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c' : E}{\Gamma, \mathsf{region}, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c\,;\, c' : E}$$

While

$$\frac{\vec{k_v} \vdash e : k \qquad \Gamma, \mathsf{region}, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c : E \qquad \forall n' \in \mathsf{sregion}(n, \emptyset).k \leq E(n')}{\Gamma, \mathsf{region}, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash [\mathsf{while}\ e\ \mathsf{do}\ c]^n : E}$$

Cond

$$\frac{\begin{array}{c}\vec{k_v} \vdash e : k \\ \Gamma, \mathsf{region}, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c : E \qquad \Gamma, \mathsf{region}, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c' : E \qquad \forall n' \in \mathsf{sregion}(n, \emptyset).k \leq E(n')\end{array}}{\Gamma, \mathsf{region}, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash [\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c']^n : E}$$

Assign2

$$\frac{\begin{array}{c}\vec{k_v} \vdash e : k \\ E(n) \sqcup \mathsf{ft}(f) \sqcup k \leq \vec{k_v}(x) \qquad \forall n' \in \mathsf{sregion}(n, \mathbf{np}).k \leq E(n') \qquad \mathsf{sHandler}(\mathsf{n}, \mathbf{np}) \uparrow \Rightarrow k \leq \vec{k_r}[\mathbf{np}]\end{array}}{\Gamma, \mathsf{region}, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash [x = e.f]^n : E}$$

Update

$$\frac{\begin{array}{c}\vec{k_v} \vdash e : k \qquad \vec{k_v} \vdash e' : k' \\ k \sqcup k' \sqcup E(n) \sqcup k_h \leq \mathsf{ft}(f) \qquad \forall n' \in \mathsf{sregion}(n, \mathbf{np}).k \leq E(n') \qquad \mathsf{sHandler}(\mathsf{n}, \mathbf{np}) \uparrow \Rightarrow k \leq \vec{k_r}[\mathbf{np}]\end{array}}{\Gamma, \mathsf{region}, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash [e.f = e']^n : E}$$

Catch

$$\frac{\Gamma, \mathsf{region}, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c : E \qquad \Gamma, \mathsf{region}, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash c' : E \qquad E(n) \leq \vec{k_v}(x)}{\Gamma, \mathsf{region}, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash [\mathsf{try}\ c\ \mathsf{catch}\ (X\ x)\ c']^n : E}$$

Return

$$\frac{\vec{k_v} \vdash e : k \qquad E(n) \sqcup k \leq \vec{k_r}[r]}{\Gamma, \mathsf{region}, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash [\mathsf{return}\ e]^n : E}$$

Throw

$$\frac{\begin{array}{c}\vec{k_v} \vdash e : k \\ \forall \mathsf{e} \in \mathsf{classAnalysis}(n)\ \mathsf{sHandler}(n, \mathsf{e}) \uparrow \Rightarrow k \leq \vec{k_r}[\mathsf{e}] \qquad \mathsf{sHandler}(\mathsf{n}, \mathsf{e}) = n' \Rightarrow \forall n' \in \mathsf{sregion}(n, \mathsf{e}).\ k \leq E(n')\end{array}}{\Gamma, \mathsf{region}, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash [\mathsf{throw}\ e]^n : E}$$

Invoke

$$\frac{\begin{array}{c}\forall i \in [0, \mathit{length}(\vec{e}) - 1].\ \vec{k_v} \vdash \vec{e}[i] : \vec{k_1'}[i] \qquad \vec{k_v}[0] \sqcup k_h \sqcup E(n) \leq k_h' \\ \vec{k_r}[\mathsf{r}] \sqcup E(n) \leq \vec{k_v}(x) \qquad \mathsf{mt}_{m'} = \vec{k_v'} \xrightarrow{k_h'} \vec{k_r'} \\ \forall \mathsf{e} \in \mathsf{excAnalysis}(m').\ \forall j \in \mathsf{sregion}(n, \mathsf{e})\ \vec{k_v}[0] \sqcup \vec{k_r'}[\mathbf{e}] \leq E(n) \qquad \mathsf{sHandler}(n, \mathsf{e}) \uparrow \Rightarrow \vec{k_v'}[0] \sqcup \vec{k_r'}[\mathsf{e}] \leq \vec{k_r}[\mathsf{e}]\end{array}}{\Gamma, \mathsf{region}, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash [x = e.m(\vec{e})]^n : E}$$

Figure 7.5: Intermediate typing rules for high-level language commands.

We define here a successor relation between commands in the language. Following Nielson et al. [18] we define for each labeled command the *init*, *final*, and *except* labels (see Figure 7.4); the set $except(n)$ is a set of pairs (class of exception, label) where label is from which there can be an —uncaught— exception of the given class..

Given a method body SP_m, $n \mapsto n'$ denotes that command labeled n' is a successor of command labeled n in the control flow graph. We define $n \mapsto n'$ iff $(n, n') \in flow(SP_m)$. The latter is defined in two steps, which are given in Figure 7.4. Let $\mapsto^+$ (resp. $\mapsto^\star$) denote the transitive (resp. transitive and reflexive) closure of $\mapsto$.

Definition 7.3.1 (branching commands, $\mathcal{LL}^\sharp$)
The branching commands *are those of the form* $\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$, $\mathsf{while}\ e\ \mathsf{do}\ c_1$, $x = e.f$, $e.f = e$, $\mathsf{throw}\ e$, *and* $x = e.m(\vec{e})$. *The set $\mathcal{LL}^\sharp$ is all the labels of branching commands in a method body.*

For example, if the method body SP_m is

$$[\ \mathsf{if}\ x\ \mathsf{then}\ [x = x]^4\ \mathsf{else}\ [x = x]^7\ ]^2; [\mathsf{return}\ x]^9$$

then $\mathcal{LL}^\sharp$ is $\{2\}$.

The following definition is needed to define control dependence regions.

Definition 7.3.2 (inner-most handler) *Let $[c]^n$ be an exception-throwing command in SP_m. Then an* inner-most handler decomposition *for $\mathbf{e} \in except([c]^n)$ consists of contexts $C_1[-]$, $C_2[-]$, command c', and label t such that*

$$SP_m \ \equiv\ C_1[[\mathsf{try}\ C_2[[c]^n]\ \mathsf{catch}\ (\mathbf{e}\ x)\ c']^t]$$

and $C_2[-]$ does not have a try-catch *that handles exceptions of class* $\mathbf{e}$ *and that encloses $[c]^n$ in its* try *part.*

We say that t is the inner-most handler *of n for* $\mathbf{e}$ *and c' is the* handler *for exception* $\mathbf{e}$ *from c.*

For any exception class and command $[c]^n$, either there is no handler for exception $\mathbf{e}$ thrown by c, or there is a unique inner-most handler decomposition for $\mathbf{e}$.

Definition 7.3.3 (handler function) *We define a function* sHandler *as follows: for label n and class* $\mathbf{e}$ *it returns the label of the inner-most handler of n for* $\mathbf{e}$*, if there is one; otherwise* $\mathsf{sHandler}(n, \mathbf{e})$ *is undefined, denoted* $\mathsf{sHandler}(n, \mathbf{e})\uparrow$.

Example 4 (inner-most handler) *In the code below,* $\mathsf{sHandler}(n, \mathbf{np}) = t$.

$$[\mathsf{try}\ ([\mathsf{try}\ [c]^n\ \mathsf{catch}\ (\mathbf{np}\ x)\ c']^t)\ \mathsf{catch}\ (X\ y)\ c'']^{n'}; \ldots$$

Control dependence regions. Control dependence regions are used by the intermediate type system to impose constraints on commands depending on branching instructions. For example, in the high level type system, if the expression e in command

$$[\mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c']^n$$

is typable as $\vdash e : k$ then c and c' can only assign variables of level at least k. The intermediate system expresses this constraint by $\forall n' \in \mathsf{sregion}(n).k \leq E(n')$, which means that every command in the region of n (including commands in branches depending on exceptions) has security level at least k.

Definition 7.3.4 (sregion and $\rhd$) *The region of a labeled command $[c]^n$ with $n \in \mathcal{LL}^\sharp$ is written* $\mathsf{sregion}(n, \mathsf{X})$ *(where* X *is and exception class if $[c]^n$ is an* X *exception-throwing command or $\emptyset$ otherwise) and defined to be the set of all labels n' such that $n \rhd n'$. Here $\rhd$ is defined inductively as follows.*

- *If $[c]^n$ is an* e *exception-throwing command with* $\mathsf{sHandler}(n, \mathsf{e})\uparrow$ *and $n \mapsto^+ n'$ then $n \rhd n'$.*

- *If* $[c]^n$ *is an* e *exception-throwing command with an inner-most handler decomposition*

$$C_1[[\mathsf{try}\ C_2[[c]^n]\ \mathsf{catch}\ (\mathsf{e}\ x)\ c']^t]$$

then there are two sub-cases:

(1) if $n \mapsto^+ n'$, $n' \in \mathsf{labels}(C_2[[c]^n])$ *and* n' *is not in the catch part of a try-catch command then* $n \rhd n'$*;*

(2) if $d \in \mathsf{labels}(c') \cup \{t\}$ *then* $n \rhd d$*. Furthermore, if* $n \rhd d$ *and* $d \rhd d'$ *then* $n \rhd d'$*.*

- *If* $[c]^n$ *is of the form* $[\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2]^n$ *and* $d \in \mathsf{labels}(c_1) \cup \mathsf{labels}(c_2)$ *then* $n \rhd d$*. Furthermore, if* $n \rhd d$ *and* $d \rhd d'$ *then* $n \rhd d'$*.*

- *If* $[c]^n$ *is of the form* $[\mathsf{while}\ e\ \mathsf{do}\ c_1]^n$ *and* $d \in \mathsf{labels}(c_1)$ *then* $n \rhd d$*. Furthermore, if* $n \rhd d$ *and* $d \rhd d'$ *then* $n \rhd d'$*.*

Note that any label in the region of an exception-throwing command is either inside the try part of its inner-most handler, or is a successor of the code in the catch part.

7.4 Connecting the high level and intermediate type systems

This section shows that if labeled source program SP is typable in the high level system then it is typable in the intermediate system as well. In order to do this, we first show how a security environment E for SP can be obtained from the typing derivation in the high level system.

Let D be a typing derivation for SP_m in the high level type system. For each constituent c of SP_m there is an instance of an introduction rule with conclusion $\Gamma, \mathsf{sregion}, \mathsf{sgn} \vdash c : k, \vec{k'}$ for some $k, \vec{k'}$ (as opposed to uses of the subsumption rule). In this case we say the type $k, \vec{k'}$ of c is *given by its intro judgment* and write $D ::\vdash c : k, \vec{k'}$. When the types $k, \vec{k'}$ are irrelevant, we use $D ::\vdash c$ to mean that SP_m is typable with derivation tree D and c occurs in SP.

The intro judgment for a subcommand reflects the essential constraints for security of this command in its context. Subsumption serves to weaken typing information, e.g., to match the two branches of a conditional in high level rule COND, but it loses precision. So we define the security environment E in terms of intro judgments.

Definition 7.4.1 (environment E from high level typing) *Let* D *be a typing derivation for source code* SP_m *in the high level type system. Define security environment* $E : \mathsf{labels}(SP_m) \to \mathcal{S}$ *as follows:*

- *If* n *belongs to some region of a branching label* n' *of a command* c' *in* SP_m *such that the intro judgment for* c' *types it with write effect or some exception effect* k' *(where* $\mathbf{e} : k' \in \vec{k'}$ *if* k' *is an exception effect and* $k' \not\leq k_{obs}$*), then* $E(n)$ *is defined as the write level of the intro judgment for* $[c]^n$ *in* D*. That is, if* $D ::\vdash c : k, \vec{k'}$ *then* $E(n) = k$*.*

- *Otherwise,* $E(n) = L$*.*

Example 5 (obtaining E from D) *Let the source code* c *be*

$$[\mathsf{if}\ y_H = 0\ \mathsf{then}\ [y_H = x_L]^6\ \mathsf{else}\ [y_H := 1]^9\,]^4;\ [\mathsf{new}\ C.f_L = 3]^{12}$$

Labels are chosen for compatibility with compilation, that will be defined in later sections. Let $\vec{k_v}(x_L) = L$*,* $\vec{k_v}(y_H) = H$ *and* $\mathsf{ft}(f_L) = L$*. The type for* c *is* L, L*. The derivation tree in the high level system shown in Fig. 5. The security environment is* $E(4) = H$*,* $E(6) = H$*,* $E(9) = H$*, and* $E(12) = L$*.*

Lemma 20 states a relation between types of commands in the high level type systems and regions.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vec{k_v}(y_H)=H}{y_H : H}\quad \dfrac{}{0:L}}{y_H = 0 : H}\quad \dfrac{\dfrac{\dfrac{\vec{k_v}(x_L)=L}{x_L : L}}{x_L : H}\quad \vec{k_v}(y_H)=H}{y_H = x_L : H\ ,\emptyset}\quad \dfrac{\dfrac{}{1:L}\quad \vec{k_v}(y_H)=H}{y_H = 1 : H\ ,\emptyset}}{\text{if } y_H = 0 \text{ then } y_H = x_L \text{ else } y_H := 1 : H\ , \mathbf{np} : L}}{\text{if } y_H = 0 \text{ then } y_H = x_L \text{ else } y_H := 1 : L\ , \mathbf{np} : L}\quad \dfrac{\dfrac{}{\text{new } C : L}\quad \text{ft}(f_L)=L\quad \dfrac{}{3:L}}{\text{new } C.f_L = 3 : L,\ \mathbf{np} : L}}{\vdash \text{if } y_H = 0 \text{ then } y_H = x_L \text{ else } y_H := 1\,;\ \text{new } C.f_L = 3 :\ L,\ \mathbf{np} : L}$$

Figure 7.6: Derivation for Example 5, using high level rules.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vec{k_v}(y_H)=H}{y_H : H}\quad \dfrac{}{0:L}}{y_H = 0 : H}\quad \dfrac{\dfrac{\vec{k_v}(x_L)=L}{x_L : L}}{x_L : H}}{y_H = x_L : E}\quad \dfrac{\dfrac{}{1:L}\quad E(9)\leq H}{[y_H = 1]^9 : E}\quad E =\uparrow_H (E, \text{sregion}(4))}{[\text{if } y_H = 0 \text{ then } y_H = x_L \text{ else } y_H := 1]^4 : E}\quad \dfrac{\begin{array}{c} 3 : L \\ \text{new } C : L \\ E(12) \leq \text{ft}(f_L) \\ E =\uparrow_L (E, \text{sregion}(12)) \\ \text{sHandler}(12) \uparrow\Rightarrow E(12) = L \end{array}}{[\text{new } C.f_L = 3]^{12} :\ E}}{\vec{k_v} \ \vdash\ [\text{if } y_H = 0 \text{ then } [y_H = x_L]^6 \text{ else } [y_H := 1]^9\,]^4;\ [\text{new } C.f_L = 3]^{12} :\ E}$$

Figure 7.7: Derivation for Example 6, using intermediate rules.

Lemma 20 *Let $[c]^n$ be an* $\mathbf{e}$ *exception-throwing command. Let $n' \in \text{sregion}(n, \mathbf{e})$ and let $D ::\vdash\ [c']^{n'} :\ k'_1, \vec{k'_2}$ be the intro judgment for n' in derivation D of the method body SP_m and let $D ::\vdash [c]^n : k_1, \vec{k_2}$ be the intro judgment for $[c]^n$. Then $k_2[\mathbf{e}] \leq k'_1$.*

The following lemma claims that the security enviromment for a command, is an upper bound for its exception effect.

Lemma 21 (exception effect and region) *Let c be an exception-throwing command. Let $D ::\vdash\ [c]^n : k, \vec{k'}$ (typable according to the corresponding intro judgment for c) and E be the security environment derived from D then for all $\mathbf{e} : k'' \in \vec{k'}$, $\forall n' \in \text{sregion}(n, \text{e})$.*

$$\vec{k''}[\text{e}] \leq E(n')$$

Now we can prove, by induction on the structure of source commands, that every program typable in the high level system is also typable in the intermediate system.

Theorem 7.4.2 *If $D ::\vdash\ c$ then $\vdash\ c : E$, where E is obtained from D by Definition 7.4.1.*

Example 6 (Preservation of types) *Recall command c of Example 5, its type derivation and its derived E. According to Definition 7.3.4, $\text{sregion}(4, \emptyset) = \{6, 9\}$. The derivation tree using the intermediate type system and E from Example 5 is given in Figure 6. Constraints $H \leq E(6), E(9)$ are satisfied since $E(6) = H$ and $E(9) = H$.*

7.5 Target language

We recall in Figure 7.8 the type system of Task 2.1 and [6].

$$\frac{P_m[i] = \mathsf{ifeq}\ j \qquad \forall j' \in \mathsf{region}(i,\emptyset),\ k \leq se(j')}{\Gamma, \mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\emptyset} k :: st \Rightarrow \mathrm{lift}_k(st)}$$

$$\frac{P_m[i] = \mathsf{return} \qquad k \sqcup se(i) \leq \vec{k_r}[n]}{\Gamma, \mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\emptyset} k :: st \Rightarrow}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \Gamma_{m_{\mathrm{ID}}}[k] = \vec{k_a'} \xrightarrow{k_h'} \vec{k_r'} \\ k \sqcup k_h \sqcup se(i) \leq k_h' \qquad \mathsf{length}(st_1) = \mathsf{nbArguments}(m_{\mathrm{ID}}) \\ k \leq \vec{k_a'}[0] \qquad \forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leq \vec{k_a'}[i+1] \\ \forall e \in \mathsf{excAnalysis}(m_{\mathrm{ID}}) \forall j \in \mathsf{region}(i,e),\ k \sqcup \vec{k_r'}[e] \leq se(j) \\ k_e = \bigsqcup \left\{ \vec{k_r'}[e] \mid \exists e \in \mathsf{excAnalysis}(m_{\mathrm{ID}}), \exists t \in \mathcal{P}_c, \mathsf{Handler}(i,e) = t \right\} \end{array}}{\Gamma, \mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\emptyset} st_1 :: k :: st_2 \Rightarrow \mathrm{lift}_{k \sqcup k_e} \left((\vec{k_r'}[n] \sqcup se(i)) :: st_2 \right)}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \Gamma_{m_{\mathrm{ID}}}[k] = \vec{k_a'} \xrightarrow{k_h'} \vec{k_r'} \\ k \sqcup k_h \sqcup se(i) \leq k_h' \qquad \mathsf{length}(st_1) = \mathsf{nbArguments}(m_{\mathrm{ID}}) \\ k \leq \vec{k_a'}[0] \qquad \forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leq \vec{k_a'}[i+1] \\ e \in \mathsf{excAnalysis}(m_{\mathrm{ID}}) \qquad \forall j \in \mathsf{region}(i,e),\ k \sqcup \vec{k_r'}[e] \leq se(j) \qquad \mathsf{Handler}(i,e) = t \end{array}}{\Gamma, \mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{e} st_1 :: k :: st_2 \Rightarrow (k \sqcup \vec{k_r'}[e]) :: \varepsilon}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{invokevirtual}\ m_{\mathrm{ID}} \qquad \Gamma_{m_{\mathrm{ID}}}[k] = \vec{k_a'} \xrightarrow{k_h'} \vec{k_r'} \\ k \sqcup k_h \sqcup se(i) \leq k_h' \qquad \mathsf{length}(st_1) = \mathsf{nbArguments}(m_{\mathrm{ID}}) \\ k \leq \vec{k_a'}[0] \qquad \forall i \in [0, \mathsf{length}(st_1) - 1],\ st_1[i] \leq \vec{k_a'}[i+1] \\ e \in \mathsf{excAnalysis}(m_{\mathrm{ID}}) \qquad k \sqcup \vec{k_r'}[e] \leq \vec{k_r}[e] \qquad \forall j \in \mathsf{region}(i,e),\ k \sqcup \vec{k_r'}[e] \leq se(j) \qquad \mathsf{Handler}(i,e) \uparrow \end{array}}{\Gamma, \mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{e} st_1 :: k :: st_2 \Rightarrow}$$

$$\frac{\begin{array}{c} P[i] = \mathsf{putfield}\ f \qquad k_1 \sqcup se(i) \sqcup k_2 \leq \mathsf{ft}(f) \qquad k_h \leq \mathsf{ft}(f) \\ \forall j \in \mathsf{region}(i,\emptyset),\ k_2 \leq se(j) \end{array}}{\Gamma, \mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\emptyset} k_1 :: k_2 :: st \Rightarrow \mathsf{lift}_{k_2} st}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{putfield}\ f \qquad k_1 \sqcup se(i) \sqcup k_2 \leq \mathsf{ft}(f) \\ \forall j \in \mathsf{region}(i,\mathbf{np}),\ k_2 \leq se(j) \qquad \mathsf{Handler}(i,\mathbf{np}) = t \end{array}}{\Gamma, \mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\mathbf{np}} k_1 :: k_2 :: st \Rightarrow k_2 \sqcup se(i) :: \epsilon}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{putfield}\ f \qquad k_1 \sqcup se(i) \sqcup k_2 \leq \mathsf{ft}(f) \\ k_2 \leq \vec{k_r}[\mathbf{np}] \qquad \forall j \in \mathsf{region}(i,\mathbf{np}),\ k_2 \leq se(j) \qquad \mathsf{Handler}(i,\mathbf{np}) \uparrow \end{array}}{\Gamma, \mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\mathbf{np}} k_1 :: k_2 :: st \Rightarrow}$$

$$\frac{P_m[i] = \mathsf{getfield}\ f \qquad \forall j \in \mathsf{region}(i,\emptyset),\ k \leq se(j)}{\Gamma, \mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\emptyset} k :: st \Rightarrow \mathrm{lift}_k((\mathsf{ft}(f) \sqcup se(i)) :: st)}$$

$$\frac{P_m[i] = \mathsf{getfield}\ f \qquad \forall j \in \mathsf{region}(i,\mathbf{np}),\ k \leq se(j) \qquad \mathsf{Handler}(i,\mathbf{np}) = t}{\Gamma, \mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\mathbf{np}} k :: st \Rightarrow k \sqcup se(i) :: \epsilon}$$

$$\frac{P_m[i] = \mathsf{getfield}\ f \qquad \mathsf{Handler}(i,\mathbf{np}) \uparrow \qquad k \leq \vec{k_r}[\mathbf{np}]}{\Gamma, \mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{\mathbf{np}} k :: st \Rightarrow}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{throw} \qquad e \in \mathsf{classAnalysis}(i) \cup \{\mathbf{np}\} \\ \forall j \in \mathsf{region}(i,e),\ k \leq se(j) \qquad \mathsf{Handler}(i,e) = t \end{array}}{\Gamma, \mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{e} k :: st \Rightarrow k \sqcup se(i) :: \epsilon}$$

$$\frac{\begin{array}{c} P_m[i] = \mathsf{throw} \qquad e \in \mathsf{classAnalysis}(i) \cup \{\mathbf{np}\} \\ k \leq \vec{k_r}[e] \qquad \forall j \in \mathsf{region}(i,e),\ k \leq se(j) \qquad \mathsf{Handler}(i,e) \uparrow \end{array}}{\Gamma, \mathsf{region}, se, \vec{k_a} \xrightarrow{k_h} \vec{k_r}, i \vdash^{e} k :: st \Rightarrow}$$

Figure 7.8: TRANSFER RULES FOR INSTRUCTIONS OF THE TARGET LANGUAGE

$$
\begin{array}{rcl}
\mathcal{E}(x) & = & \mathsf{load}\ x \\
\mathcal{E}(n) & = & \mathsf{push}\ n \\
\mathcal{E}(e\ op\ e') & = & \mathcal{E}(e) :: \mathcal{E}(e') :: \mathsf{binop}\ op \\
\mathcal{S}(x = e.f) & = & \mathcal{E}(e) :: \underline{\mathsf{getfield}\ f} :: \mathsf{store}\ x \\
\mathcal{E}(\mathsf{new}\ C) & = & \mathsf{new}\ C \\
\mathcal{S}(x = e) & = & \mathcal{E}(e) :: \underline{\mathsf{store}}\ x \\
\mathcal{S}(c_1;\ c_2) & = & \mathcal{S}(c_1) :: \mathcal{S}(c_2) \\
\mathcal{S}(\mathsf{while}\ e\ \mathsf{do}\ c) & = & \quad let \quad le = \mathcal{E}(e); lc = \mathcal{S}(c); \\
 & & \quad in \quad \mathsf{goto}\ (pc + \#lc + 1) :: lc :: le :: \underline{\mathsf{ifeq}\ (pc - \#lc - \#le)} \\
\mathcal{S}(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) & = & \quad let \quad le = \mathcal{E}(e); lc_1 = \mathcal{S}(c_1); lc_2 = \mathcal{S}(c_2); \\
 & & \quad in \quad le :: \underline{\mathsf{ifeq}\ (pc + \#lc_2 + 2)} :: lc_2 :: \mathsf{goto}\ (pc + \#lc_1 + 1) :: lc_1 \\
\mathcal{S}(e.f = e') & = & \mathcal{E}(e') :: \mathcal{E}(e) :: \underline{\mathsf{putfield}\ f} \\
\mathcal{S}(\mathsf{throw}\ e) & = & \mathcal{E}(e) :: \underline{\mathsf{throw}} \\
\mathcal{S}(\mathsf{try}\ c_1\ \mathsf{catch}\ (X\ x)\ c_2) & = & \quad let \quad lc_1 = \mathcal{S}(c_1); lc_2 = \mathcal{S}(c_2); \\
 & & \quad in \quad lc_1 :: \mathsf{goto}\ (pc + \#lc_2 + 1) :: \underline{\mathsf{store}\ x} :: lc_2 \\
\mathcal{S}(\mathsf{return}\ e) & = & \mathcal{E}(e) :: \mathsf{return} \\
\mathcal{S}(x = e.m(\vec{e})) & = & \mathcal{E}(\vec{e}) :: \mathcal{E}(e) :: \underline{\mathsf{invoke}\ m}
\end{array}
$$

Figure 7.9: Compilation function. Primary instructions are underlined.

$$
\begin{array}{rcl}
\mathcal{X}(c_1;\ c_2) & = & \mathcal{X}(c_1) :: \mathcal{X}(c_2) \\
\mathcal{X}(\mathsf{while}\ e\ \mathsf{do}\ c) & = & \mathcal{X}(c) \\
\mathcal{X}(\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2) & = & \mathcal{X}(c_1) :: \mathcal{X}(c_2); \\
\mathcal{X}(\mathsf{try}\ c_1\ \mathsf{catch}\ (X\ y)\ c_2) & = & \quad let \quad lc_1 = \mathcal{S}(c_1); lc_2 = \mathcal{S}(c_2); \\
 & & \quad in \quad \mathcal{X}(c_1) :: \mathcal{X}(c_2) :: \langle 1, \#lc_1 + 1, \#lc_1 + 2, X \rangle \\
\mathcal{X}(_) & = & \epsilon
\end{array}
$$

Figure 7.10: Definition of exception table.

7.6 Compilation.

Compilation is done by a function, $\mathcal{W}$, from source programs to target programs. The compiler is based on [40].

The compilation function from source programs to target programs $\mathcal{W} : \mathsf{Prog} \to \mathsf{Prog}_c$ is defined from a compilation function on expressions $\mathcal{E} : \mathsf{Expr} \to \mathsf{Instr}^\star$, and a compilation function on commands $\mathcal{S} : \mathsf{Comm} \to \mathsf{Instr}^\star$. Their formal definitions are given in Figure 7.9. The compilation of every labeled command includes a *primary instruction* —e.g., getfield is the primary instruction for a field access $[x = e.f]^n$— and these are indicated in the definition of the compiler. Exception tables are defined in Figure 7.10. In order to enhance readability, we use :: both for consing an element to a list and concatenating two lists, and we omit details of calculating program points (we use pc to represent current program point) in the clauses for while and if expressions. We also use # to denote the length of a list.

For method body $SP_m \equiv c; \mathsf{return}\ e$, we define the compilation $\mathcal{W}(SP_m)$ to be $\mathcal{S}(c; \mathsf{return}\ e)$.

We assume *label compatibility*: the label of a source command is the same as the label of the program point of the primary instruction in its compilation, e.g., if getfield is obtained by compilation of $[x = e.f]^n$, its corresponding program point in the target program is n. This implies that $\mathcal{LL}^\sharp = \mathcal{PP}^\sharp$ for a source

program and its compilation. The simplifying assumption loses no generality since source labels can be read off the compilation (as was done in the example just after Definition 7.3.1).

Given the definition of regions for the source program, we obtain regions for its compilation as follows.

Definition 7.6.1 (compiler for regions) *Let* $[c']^{n'}$ *be a branching source command. Let* $\tau \in \{\emptyset\} + \mathcal{C}$. *Then define* $\mathsf{tregion}(n', \tau)$ *to be the union, over all* $[c]^n$ *with* $n \in \mathsf{sregion}(n', \tau)$ *(in the source language), of* $\{i..j\}$ *where* $TP[i..j]$ *is the compilation of* c. *That is, all program points in the compilation of commands in the (source) region of some branching command with label* n' *are included in the (target) region of the compiled program for the branching instruction* n'.

Furthermore

- *if* n' *is a command of the form* $\mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2$, *and* $\#lc_2$ *is the length of the compilation of command* c_2, *then* $n' + lc_2 + 1 \in \mathsf{tregion}(n, \emptyset)$ *(note that* $n' + lc_2 + 1$ *corresponds to the* goto *instruction).*
- *if* n' *is an* $\mathbf{e}$ *exception-throwing command with* $\mathsf{sHandler}(n', \mathbf{e}) = t$, *then program points* t *and* $t - 1$ *corresponding to* goto *and* store *instructions in the compilation of a try-catch with label* t, *are also included in region of* n'.
- *if* n *is a while command, then* $n \in \mathsf{tregion}(n, \emptyset)$.

We will use regions for the target language defined as in Definition 7.6.1 for proofs of preservation, in next section. The following lemma claims that regions defined as in Definition 7.6.1 have the SOAP property.

Lemma 22 *Let* $n \in \mathcal{PP}^\sharp$ *be a program point in a target program* P *and let* $\mathsf{tregion}(n)$ *be defined as in Definition 7.6.1 from compilation of a command* $[c]^n$. *Then SOAP holds for* $\mathsf{tregion}(n)$.

7.7 Connecting the intermediate and target type systems

The main result is that compilation preserves typing. That is, given a typing derivation for a source program in the high level system, a corresponding security type for bytecode can be obtained. We show that the defined security type satisfies all the conditions for typing of a bytecode program.

First, given a source program SP together with a security environment E for it we obtain a security environment se with which to type the compilation of SP.

Definition 7.7.1 (se **determined by** E**)** *We define se by induction on syntax of source commands. The domain of se is the set of program points in the compilation* $\mathcal{W}(SP_m)$. *Define* $se(i)$ *as* $E(n)$ *where* $[c]^n$ *is the smallest subcommand of* SP_m *whose compilation contains program point* i.

Lemma 23 *Let* c *be an exception throwing command. Suppose* $D :: \vdash [c]^n : k, k'$ *(intro judgment), let* E *be the security environment derived from* D, *and let* se *be determined by* E. *Then for all* $\mathsf{e} : k'' \in \vec{k'}$, $\forall n' \in \mathsf{tregion}(n, \mathbf{e}). k'' \leq se(n')$.

Main result. Finally we can show that compilation of expressions and of commands preserves typing. From these two lemmas we obtain the main theorem.

Lemma 24 *Let* e *be an expression in* $[c]^n$ *such that* $[c]^n$ *is the inner-most command that encloses* e *and* $\vec{k_v} \vdash c :\ E$ *and* $\vec{k_v} \vdash e : k$, *and* $\mathcal{S}(c)[i..j] = \mathcal{E}(e)$. *Let se be the security environment determined by* E. *Then for any* $st_i \in \mathcal{ST}$ *there exist* $st_{i+1}, ..st_j$ *such that the following hold:*

1. *for every* $l \mapsto^{\emptyset} l'$ *in* $i..j$ *then* $\Gamma, \mathsf{tregion}, se, \mathsf{sgn}, l \vdash st_l \Rightarrow st_{l'}$;
2. $\Gamma, \mathsf{tregion}, se, \mathsf{sgn}, j \vdash st_j \Rightarrow (k \sqcup se(i)) :: st_i$.

In the following, for simplicity's sake we fix Γ, se, and tregion and we abbreviate $\Gamma, \mathsf{tregion}, se, i \vdash st \Rightarrow st'$ as $i \vdash st \Rightarrow st'$.

The following lemma claims that for every typable compiled command, there exist types in the target type system.

Lemma 25 *Let c be a command in source method body SP_m, typed $\vdash c : E$, where E is obtained as in Definition 5, and let $[i..j]$ be the program points in compilation of c. Let se be the security environment determined by E and let* tregion *be defined as in Definition 7.6.1. Then, for all st there exists $st_{i+1}, .., st_j$ such that if we define $st_i = st$, we have*

- $\Gamma, \mathsf{region}, \vec{k_v} \xrightarrow{k_h} \vec{k_r}, l \vdash st_l \Rightarrow st_{l'}$, *for all $l \mapsto l'$, $l \in \{i..j\}$;*
- *if c is not a throw command, then $st_j = st$;*
- *if $[c]^n$ is a* e *exception-throwing command with* $\mathsf{sHandler}(n, \mathsf{e}) = n'$, *then $n \vdash st_n \Rightarrow k' :: st$, with $k' \leq se(n')$*

Finally, putting together Lemma 25 and Theorem 7.4.2 we obtain the main result.

Theorem 7.7.2 *Let $\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash SP_m$, i.e. source program SP_m be typable in the high level system. Then $\Gamma, \vec{k_v} \xrightarrow{k_h} \vec{k_r} \vdash \mathcal{W}(SP)$, i.e., its compilation is typable in the target system.*

Example 7 *The compilation $\mathcal{S}(c)$ of the program introduced in Example 5 and its types obtained with the Target type system is shown below. To construct se for this program, we use E from Example 5 and Definition 7.7.1 to obtain: $se = \{1 \mapsto H, 2 \mapsto H, 3 \mapsto H, 4 \mapsto H, 5 \mapsto H, 6 \mapsto H, 7 \mapsto H, 8 \mapsto H, 9 \mapsto H, 10 \mapsto L, 11 \mapsto L, 12 \mapsto L\}$.*

1	load y_H	ϵ
2	push 0	$H :: \epsilon$
3	binop $=$	$H :: H :: \epsilon$
4	ifeq 8	$H :: \epsilon$
5	load x_L	ϵ
6	store y_H	$H :: \epsilon$
7	goto 10	ϵ
8	push 1	ϵ
9	store y_H	$H :: \epsilon$
10	new C	ϵ
11	push 3	$L :: \epsilon$
12	putfield f_L	$L :: L$
		ϵ

It is easy to corroborate that the constraints of the type system to apply the transfer rules with the types given above hold.

7.8 Discussion

The development and deployment of software that respects an end-to-end confidentiality policy requires a number of ingredients. First, developers must be able to specify how interfaces relate with the policy using labels. They need tools to provide them accurate feedback whether their labeling is consistent with the policy. Second, execution platforms must reflect the assumptions of the policy (e.g. a low attacker is not able to directly read channels labeled high), and must feature security functions that enforce the policy.

Verification tools for developers and security functions for execution platforms have a similar purpose, namely to verify that the labeling is correct and ensures the policy. Establishing a formal relation between them, and devising means to exploit the results of source code verification for verification of executables is a significant step towards the adoption of end-to-end security policies in mobile code.

The focus of this chapter is on checking that a program is noninterferent with respect to a given labeling—that is, controlling fine-grained flows within a program.

We are confident that our approach is robust and can be adapted to more sophisticated settings, including richer languages, more expressive policies, and optimizing compilers.

Richer languages. Our language already handles some main complexities of Java, including exceptions and heap allocated mutable objects. The full version of the article also considers methods and multiple exceptions. We expect to treat multiple exceptions in a way similar to JFlow (corresponding to our single exception effect, JFlow uses a list of levels indexed by exception types, called "paths"). We believe that type preservation can be extended to sequential Java, provided existing type systems at source code and bytecode are extended appropriately—in particular, it is possible to treat JVM subroutines, but not so interesting as they will disappear from Java 1.6. The real challenge is to understand whether type preservation scales up to concurrent Java, but for the time being there is no information flow type system that has been proved sound for a concurrent fragment of Java source code or bytecode.

More sophisticated policies. Practical end-to-end policies weaker than full non-interference are often needed, e.g., to cater for downgrading [39] and to connect flow policy with access controls (based on stack inspection or execution history) [2, 3]. Such policies are under active investigation but several current proposals provide source level type systems to express and statically enforce specific policies that enable declassification. We believe that most of these type systems can be adapted to bytecode in such a way that type-preservation will be ensured. For example, it is trivial to extend type preservation to an extension of Java with a cryptographic API where encryption turns secret data into public one.

Practical enforcement mechanisms for end-to-end policies are also likely to combine information flow type systems with other type systems, e.g. for exception analysis, or with logical verification methods. Extending type-preservation results to combinations of type systems is thus an interesting topic for future work, as is preservation of typability/provability in a framework that combines type systems and logical reasoning.

Optimizing compilers. Common Java compilers such as Sun's javac or IBM's jikes only perform very limited optimizations such as constant folding, dead code elimination, and rewriting conditionals whose conditions always evaluate to the same constant. These source-to-source transformations can easily be shown to preserve information-flow typing; thus type preservation lifts to standard Java compilers.

More aggressive optimizations may break type preservation, even though they are semantics preserving, and therefor security preserving. Of course, if the transformations are made after bytecode verification (as in JIT), type preservation is not needed (instead transformations become part of the TCB). For example, applying common subexpression elimination to the program $x_H := n_1 * n_2; y_L := n_1 * n_2$, where n_1 and n_2 are constant values, will result into the program $x_H := n_1 * n_2; y_L := x_H$. Assuming that variable x_H is a high variable and y_L is a low variable, the original program is typable, but the optimized program is not, since the typing rule for assignment will detect an explicit flow $y_L := x_H$. (A naive solution to recover typability is to create a low auxiliary variable z_L in which to store the result of the computation $n_1 * n_2$, and assign z_L to x_H and y_L, i.e. $z_L := n_1 * n_2; x_H := z_L; y_L := z_L$.) Adapting standard program optimizations so that they do not break type preservation is left as future work. We conjecture that most optimizations will only need minor modifications, provided advanced features of the type system such as label polymorphism are available.

Chapter 8

Conclusions and future work

In this deliverable, we presented the main ideas of both proof-transforming and type-preserving compilation. Both techniques enable reasoning about interesting security properties on source code and using the result directly on bytecode to build a certificate that can be checked efficiently.

8.1 Proof-transforming compilation

We designed a new direct verification condition generator and integrated it into the MOBIUS PVE, which is being built in Task 3.6. The new VCGen is tailored towards the already existing verification condition generator for bytecode and enables automatic transformation of proofs to the bytecode level. The verification condition generation process can be separated into two relatively independent steps.

First, JML annotations are transformed to first order logic specification tables from which the actual VCGen retrieves pre- and postconditions for routines as well as local annotations for program statements. With the prototypical implementation, we are able to process all Java constructs except multi-dimensional arrays and concurrency instructions. On the JML side, we currently support most interesting JML-level 0 constructs except for model fields, array handling instructions, and some visibility modifiers.

Second, we also showed that it is possible to achieve nearly preservation of proof obligations for the chosen language subset, which means that translating a proof from source code to bytecode is feasible. This is done by generating structurally identical verification conditions on source and bytecode that only differ in minor aspects like variable names or the change of boolean variables on source code to integer variables on bytecode. The proofs for the VCs on source and bytecode therefore are very similar and can be transformed from one to the other. This part sets the important theoretical foundations for the proof transformer implementation and shows that the whole idea of proof-transforming compilation is effectively doable for a realistic language subset.

The architecture of the proof-transforming compiler has been chosen so that the trusted code base is as small as possible. No part of the tool-chain needs to be trusted but only Coq as proof checker and Bicolano as the JVM model.

With the MOBIUS base logic in mind, we also succeeded to define a proof transformer that uses a Hoare logic. The main difficulty is that the proof contains the structure of the program and, therefore, the translation is more complex. Still, we could prove that the transformed proof is correct if the original proof is correct.

Future work: The main work for the next part of Task 4.4 will consist of the implementation of the proof transformer, for which we already set the theoretical foundations. Another important task is the extension of the proof-transforming compiler to work with optimizations on bytecode. The key idea is to first use the non-optimizing proof-transforming compiler and then optimize the resulting bytecode along with rewriting the proof such that it still holds for the optimized version.

8.2 Type-preserving compilation

The focus of the type-preserving compilation chapter of this deliverable is on checking that a program is noninterferent with respect to a given labeling—that is, controlling fine-grained flows within a program. The chapter follows a long line of work in this area and makes significant progress by showing a formal relation between typability at source code, and bytecode verification for an extended bytecode verifier that enforces noninterference.

In fact, we obtained this relation by deriving the typing systems for source code from the bytecode system. First, we establish a correspondence between regions in source code and regions in bytecode. Then we obtain constraints on a security environment for source fragment c, in terms of regions, by combining the constraints from the bytecode rules as applied to the compilation of c. Next, we formulate high level judgements and a connection between them and the security environment. Finally, this connection is used to obtain a high level rule for each source code construct, using the constraints in the construct's intermediate rule and the definition of regions for source code.

We have solved the problem of connecting control dependence between source and target language—even reducing control dependence to an inductive property amenable to machine verification—and we are confident that our approach is robust and can be adapted to more sophisticated settings, including richer languages, more sophisticated policies, and optimizing compilers.

Future work: The work on type-preserving compilation is going to be extended in two directions. On the one hand, the concept of type preserving compilation is going to be extended to distributed programs. On the other hand, we will study preservation of information-flow types in optimizations, similar to our work on transforming proofs in optimizations.

Bibliography

[1] M. Abadi. Secrecy by typing in security protocols. *Journal of the Association of Computing Machinery*, 46(5):749–786, September 1999.

[2] A. Banerjee and D. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15:131–177, March 2005. Special Issue on Language-Based Security.

[3] A. Banerjee and D. A. Naumann. History-based access control and secure information flow. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Proceedings of CASSIS'04*, volume 3362 of *Lecture Notes in Computer Science*, pages 27–48. Springer-Verlag, 2004.

[4] F. Y. Bannwart and P. Müller. A program logic for bytecode. *Electronic Notes in Theoretical Computer Science*, 141:255–273, 2005.

[5] G. Barthe, L. Burdy, J. Charles, B. Grégoire, M. Huisman, J.-L. Lanet, M. Pavlova, and A. Requet. JACK: A tool for validation of security and behaviour of Java applications. In *Formal Methods for Components and Objects: Revised Lectures from the 5th International Symposium FMCO 2006*, number 4709 in Lecture Notes in Computer Science, pages 152–174. Springer-Verlag, 2007.

[6] G. Barthe, D. Pichardie, and T. Rezk. A certified lightweight non-interference Java bytecode verifier. In *Programming Languages and Systems: Proceedings of the 16th European Symposium on Programming, ESOP 2007*, number 4421 in Lecture Notes in Computer Science, pages 125–140. Springer-Verlag, 2007.

[7] G. Barthe and T. Rezk. Non-interference for a JVM-like language. In M. Fähndrich, editor, *Types in Language Design and Implementation*, pages 103–112. ACM Press, 2005.

[8] G. Barthe, T. Rezk, and A. Saabas. Proof obligations preserving compilation. In T. Dimitrakos, F. Martinelli, P. Ryan, and S. Schneider, editors, *Workshop on Formal Aspects in Security and Trust*, number 3866 in Lecture Notes in Computer Science, pages 112–126. Springer-Verlag, 2005.

[9] G. Boudol and I. Castellani. Noninterference for concurrent programs and thread systems. In *Theoretical Computer Science*, volume 281, pages 109–130, 2002.

[10] D. Cok and J. R. Kiniry. ESC/Java2: Uniting ESC/Java and JML: Progress and issues in building and using ESC/Java2 and a report on a case study involving the use of ESC/Java2 to verify portions of an internet voting tally system. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure and Interoperable Smart Devices: Proceedings of the International Workshop CASSIS 2004*, volume 3362 of *Lecture Notes in Computer Science*, pages 108–128. Springer-Verlag, 2005.

[11] C. Colby, P. Lee, G. C. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for Java. In *Programming Languages Design and Implementation*, volume 35 of *ACM Sigplan Notices*, pages 95–107. ACM Press, June 2000.

[12] Á. Darvas and P. Müller. Formal encoding of JML level 0 specifications in JIVE. Technical report, ETH Zurich, 2007. Annual Report of the Chair of Software Engineering.

[13] K. K. Dhara and G. T. Leavens. Weak behavioral subtyping for types with mutable objects. In S. Brookes, M. Main, A. Melton, and M. Mislove, editors, *Mathematical Foundations of Programming Semantics, Eleventh Annual Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995. Available from `http://www.sciencedirect.com/science/journal/15710661`.

[14] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, October 2005.

[15] Eclipse website. www. See http://www.eclipse.org/.

[16] ESC/Java2 website. www. See `http://secure.ucd.ie/products/opensource/ESCJava2`.

[17] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Programming Languages Design and Implementation*, volume 37, pages 234–245, June 2002.

[18] C. Hankin, F. Nielson, and H. R. Nielson. *Principles of Program Analysis*. Springer-Verlag, 2005. Second Ed.

[19] Jack website. www. See http://www-sop.inria.fr/everest/soft/Jack/jack.html.

[20] G. T. Leavens and K. K. Dhara. Concepts of behavioral subtyping and a sketch of their extension to component-based systems. pages 113–135, 2000.

[21] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, and J. Kiniry. *JML Reference Manual*, July 2005. In Progress. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`.

[22] MOBIUS Consortium. Deliverable 2.1: Intermediate report on type systems, 2006. Available online from `http://mobius.inria.fr`.

[23] MOBIUS Consortium. Deliverable 3.1: Bytecode specification language and program logic, 2006. Available online from `http://mobius.inria.fr`.

[24] MOBIUS Consortium. Deliverable 4.2: Certificates, 2007. Available online from `http://mobius.inria.fr`.

[25] G. Morrisett, D. Walker, K. Crary, and N. Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, 1999. Expanded version of a paper presented at POPL 1998.

[26] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002.

[27] P. Müller and M. Nordio. Proof-transforming compilation of programs with abrupt termination. Technical Report 565, ETH Zurich, 2007.

[28] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.

[29] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Principles of Programming Languages*, pages 228–241. ACM Press, 1999. Ongoing development at `http://www.cs.cornell.edu/jif/`.

[30] G. C. Necula. Proof-carrying code. In *Principles of Programming Languages*, pages 106–119, New York, NY, USA, 1997. ACM Press.

[31] G. C. Necula and P. Lee. The design and implementation of a certifying compiler. In *Programming Languages Design and Implementation*, volume 33, pages 333–344, New York, NY, USA, 1998. ACM Press.

[32] D. Pichardie. Bicolano – Byte Code Language in Coq. `http://mobius.inria.fr/bicolano`. Summary appears in [23], 2006.

[33] A. Poetzsch-Heffter and P. Müller. Logical foundations for typed object-oriented languages. In D. Gries and W. De Roever, editors, *Programming Concepts and Methods (PROCOMET)*, pages 404–423, 1998.

[34] A. Poetzsch-Heffter and P. Müller. A programming logic for sequential Java. In S. D. Swierstra, editor, *European Symposium on Programming Languages and Systems (ESOP'99)*, volume 1576 of *Lecture Notes in Computer Science*, pages 162–176. Springer-Verlag, 1999.

[35] A. Poetzsch-Heffter and N. Rauch. Soundness and relative completeness of a programming logic for a sequential Java subset. Technical report, Technische Universität Kaiserlautern, 2004.

[36] ProverEditor website. www. See `http://provereditor.gforge.inria.fr`.

[37] A. D. Raghavan and G. T. Leavens. Desugaring JML method specifications. Technical Report TR #00-03e, Department of Computer Science, Iowa State University, 2000. Current revision from May 2005.

[38] V. P. Ranganath, T. Amtoft, A. Banerjee, M. B. Dwyer, and J. Hatcliff. A new foundation for control-dependence and slicing for modern program structures. In Mooly Sagiv, editor, *European Symposium on Programming*, pages 77–93, 2005.

[39] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Computer Security Foundations Workshop*, pages 255–269. IEEE Press, 2005.

[40] R.F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine—Definition, Verification, Validation*. Springer-Verlag, 2001.