Proof-Transforming Compilation of Programs with Abrupt Termination

Peter Müller and Martin Nordio

ETH Zurich, Switzerland
{Peter.Mueller,Martin.Nordio}@inf.ethz.ch

ETH Technical Report 565

May 2007

Abstract

The execution of untrusted bytecode programs can produce undesired behavior. A proof on the bytecode programs can be generated to ensure safe execution. Automatic techniques to generate proofs, such as certifying compilation, can only be used for a restricted set of properties such as type safety. Interactive verification of bytecode is difficult due to its unstructured control flow. Our approach is verify programs on the source level and then translate the proof to the bytecode level. This translation is non-trivial for programs with abrupt termination. We present proof transforming compilation from Java to Java Bytecode. This paper formalizes the proof transformation and present a soundness result.

Keywords: Trusted Components, Proof-Carrying Components, Proof-Transforming Compiler.

Contents

List of Figures

1 Introduction

Proof-Carrying Code (PCC) [13, 14] has been developed with the goal of solving the problems produced by the unsafe execution of mobile code. In PCC, the code producer provides a *proof*, a certificate that the code does not violate the security properties of the code consumer. Before the code execution, the proof is checked by the code consumer. Only if the proof is correct, the code is executed.

The certificate proves the properties that are satisfied by the bytecode program. With the goal of generating certificates automatically, Necula [?] has developed certifying compilers. *Certifying compilers* are compilers that take a program as input and produce bytecode and its proof. Unfortunately, certifying compilers only work with a restricted set of provable properties such as type safety.

Another approach to solve the problem caused by mobile code is *interactive verification of bytecode*. This approach is applicable to a wide range of properties, but is difficult due to the bytecode's unstructured control flow. Contrary, source verification is simpler, but does not generate a certificate for the bytecode program.

The approach we propose here is the use of a Proof - Transforming Compiler (PTC). PTCs are similar to certifying compilers in PCC, but take a source proof as input and produce the bytecode proof. Figure 1 shows the architecture of this approach. The code producer develops a program. A proof of the source program is developed using a prover. Then, the PTC translates the proof producing the bytecode and its proof, which are sent to the code consumer. The proof checker verifies the proof. If the source proof or the translation were incorrect, the checker would reject the code.

An important property of Proof-Transforming Compilers is that they do not have to be trusted. If the compiler produces a wrong specification or a wrong proof for a component, the proof checker will reject the component. This approach has the strengths of both above mentioned approaches.

If the source and target languages are close, the proof translation is 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 not simple. Compiling these statements in isolation is simple, but the compilation of their interplay is not.

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 paper, we present the first PTC that handles these complications.

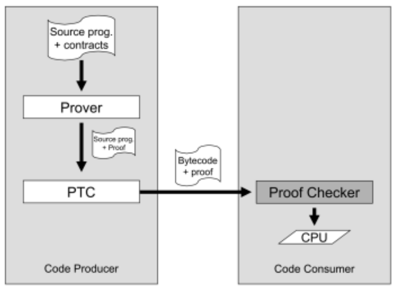

Figure 1: General architecture.

Outline. A PTC consist of five elements: a source language, a logic for the source language, a bytecode language, a logic for the bytecode and the translation functions. The source language is presented in section 2. We present the logic for COOL in section 3. We present the Bytecode language and its bytecode logic in sections 4 and 5, resp. In the section 6 we present the proof translation rules from the source language to Bytecode. Two examples of the application of while and try-catch translation are presented in section 7. The conclusions and future work are presented in section 10. Appendix A presents the soundness proof of the translation. Appendix B presents the logic and its translation for C#.

2 The source language

The source language is similar to a subset of C#, Eiffel and Java called COOL (Common Object-Oriented program Language). The COOL language includes basic statements like assign, conditional and compositional; while loops and break statements; object-oriented features such as cast, new, read and write field, and method invocation; and exception handling.

Following, we present a short description of the language constructs that are part of COOL:

- **methods:**
  - nonstatic methods with one parameter. There is no return statement. The return value of a method has to be assigned to a special variable **result**
  - abstract methods
  - nonstatic attributes
  - a list of local variable declarations at the beginning of a block
- **statements:**
  - assign statement
  - conditional statement if then else and if then
  - compositional statement
  - The object-oriented statements of COOL are cast, new object, write field, read field and invocation of methods
  - throw and break statements. Break statements are unlabelled breaks and the language does not have neither labelled break nor continue.
  - try catch statements. We assume that every try block has exactly one catch block
  - finally statement. Every try block has at most one finally block
- **expressions:**
  - Expressions are side-effect-free expressions. Furthermore, expressions cannot throw an exception.
  - Cast may not appear within expressions.

3 The logic for COOL

In this section we present the logic for the source programming language. The logic is based on the programming logic presented in [19]. We modify it and propose new rules for *break*, *try catch* and *finally* (subsection 3.4 and 3.5, resp.).

A *method implementation T@m* represents the concrete implementation of method m in class T. A *virtual method T:m* represents the common properties of all method implementations that might by invoked dynamically when m is called on a receiver of static type T, that is, *impl(T,m)* (if *T:m* is not abstract) and all overriding subclass methods.

Properties of methods and method bodies are expressed by Hoare triples of the form {P} *comp* {Q}, where P, Q are sorted first-order formulas and *comp* is a method implementation T@m, a virtual method T:m or a method body p. We call such a triple *method specification*. 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 in which Q holds, or (2) *comp* aborts due errors or actions that are beyond the semantics of the programming language (for instance, memory allocation problems), or (3) *comp* runs forever.

The state of our COOL programs consists of local variables, parameters and the object store \$. The object store models the heap. It describes the states of all objects in a program at a certain point of execution. We use the object store presented in [23]. Following we present a short list of the operation we use for the object store.

- $instvar : Value \times FieldDeclId \rightarrow InstVar$: It returns the instance variable lookup.
- $\$ < f := v >$: $ObjectStore \times InstVar \times Value \rightarrow ObjectStore$: It returns the object store after an instance variable update.
- $\$(f) : ObjectStore \times InstVar \rightarrow Value$: It returns the instance variable load.
- $\$ < T >$: $ObjectStore \times ClassTypeId \rightarrow ObjectStore$. It returns the object store after the allocation of a new object of type T.
- $new(\$, T) : ObjectStore \times ClassTypeId \rightarrow Value$: it returns a new object of type T.

$\mathcal{X}$ is a program variable used to describe the current status of the program. This is required to treat jumps and abrupt termination appropriately. The possible values of $\mathcal{X}$ are: *normal*, *break*, and *exc*. The value *break* indicates that we are currently processing a break jump. *exc* indicates that an exception has been thrown and *normal* for normal execution.

We assume that the source logic is normalized. We assume that every precondition is written as P where P consists of tree parts: the normal condition, the break condition and the exception condition. The break and the exception condition are false. And the normal condition (P_n) does not have occurrences of the variable $\mathcal{X}$. This means that the precondition is a normal precondition. Formally,

$$P \equiv \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & P_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & false) \wedge \\ (\mathcal{X} = exc & \Rightarrow & false) \end{array} \right\}$$

Also, we assume that the postcondition is written as Q where Q consists of three parts (Q_n, Q_b and Q_e resp.), where Q_n, Q_b and Q_e do not have occurrences of variable $\mathcal{X}$. Formally,

$$Q \equiv \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & Q_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & Q_b) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e) \end{array} \right\}$$

To make the logic easy to read and understand, we write the Hoare triples as:

$$\left\{ \; P_n \; \right\} \quad s_1 \quad \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & Q_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & Q_b) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e) \end{array} \right\}$$

but it means:

$$\left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & P_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & false) \wedge \\ (\mathcal{X} = exc & \Rightarrow & false) \end{array} \right\} \quad s_1 \quad \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & Q_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & Q_b) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e) \end{array} \right\}$$

3.1 Assign, conditional and compositional statements

3.1.1 Assign Statement

$$\frac{}{\left\{\ P_n[e/x]\ \right\}\quad x = e \quad \left\{\begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & P)\ \wedge \\ (\mathcal{X} = \mathit{break} & \Rightarrow & \mathit{false})\ \wedge \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & \mathit{false}) \end{array}\right\}}$$

3.1.2 Conditional Statement

$$\frac{\begin{array}{c} \left\{\ P_n\ \wedge\ e\ \right\}\quad s_1 \quad \left\{\begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q_n)\ \wedge \\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q_b)\ \wedge \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & Q_e) \end{array}\right\} \\ \\ \left\{\ P_n\ \wedge\ \neg e\ \right\}\quad s_2 \quad \left\{\begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q_n)\ \wedge \\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q_b)\ \wedge \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & Q_e) \end{array}\right\} \end{array}}{\left\{\ P_n\ \right\}\quad \mathit{if}\ (e)\ \mathit{then}\ s_1\ \mathit{else}\ s_2 \quad \left\{\begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q_n)\ \wedge \\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q_b)\ \wedge \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & Q_e) \end{array}\right\}}$$

3.1.3 Compositional Statement

In the composition statement, we need to check whether the statement has been executed normally or not. First, the statement s_1 is executed and after that we check whether it finishes in normal execution or not. We execute the statement s_2 if only if s_1 finished in normal execution.

$$\frac{\begin{array}{c} \left\{\ P_n\ \right\}\quad s_1 \quad \left\{\begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q_n)\ \wedge \\ (\mathcal{X} = \mathit{break} & \Rightarrow & R_b)\ \wedge \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & R_e) \end{array}\right\} \\ \\ \left\{\ Q_n\ \right\}\quad s_2 \quad \left\{\begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & R_n)\ \wedge \\ (\mathcal{X} = \mathit{break} & \Rightarrow & R_b)\ \wedge \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & R_e) \end{array}\right\} \end{array}}{\left\{\ P_n\ \right\}\quad s_1; s_2 \quad \left\{\begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & R_n)\ \wedge \\ (\mathcal{X} = \mathit{break} & \Rightarrow & R_b)\ \wedge \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & R_e) \end{array}\right\}}$$

3.2 Rules for object-oriented features

3.2.1 Cast

$$\left\{ \begin{array}{l} (\ \tau(e)\ \preceq\ T \wedge\ P_n[e/x]\ )\vee \\ (\ \tau(e)\ \not\preceq\ T\wedge \\ Q_e \left[\begin{array}{l} \$ < CastExc > /\$, \\ new(\$, CastExc)/excV \end{array} \right]) \end{array} \right\}\ x = (T)\ e\ \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & P) \wedge \\ (\mathcal{X} = break & \Rightarrow & false) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e) \end{array} \right\}$$

3.2.2 New Object

$$\left\{\ P_n[new(\$, T)/x, \$ < T > /\$]\ \right\}\ x = new\ T()\ \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & P_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & false) \wedge \\ (\mathcal{X} = exc & \Rightarrow & false) \end{array} \right\}$$

3.2.3 Read Field

$$\left\{ \begin{array}{l} (y \neq null\ \wedge\ P_n[\$(instvar(y, S@a))/x]) \vee \\ (y = null\ \wedge\ Q_e \left[\begin{array}{l} \$ < NullPExc > /\$, \\ new(\$, NullPExc)/excV \end{array} \right]) \\ \end{array} \right\}\ x = y.S@a\ \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & P_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & false) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e) \end{array} \right\}$$

3.2.4 Write Field

$$\left\{ \begin{array}{l} (y \neq null\ \wedge\ P_n[\$ < instvar(y, S@a) := e > /\$]) \vee \\ (y = null\ \wedge\ Q_e \left[\begin{array}{l} \$ < NullPExc > /\$, \\ new(\$, NullPExc)/excV \end{array} \right]) \\ \end{array} \right\}\ y.S@a = e\ \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & P_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & false) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e) \end{array} \right\}$$

3.2.5 Invocation

$$\frac{\left\{\ P_n\ \right\}\ T : m(p)\ \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & Q_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & false) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e) \end{array} \right\}}{\left\{ \begin{array}{l} (y \neq null\ \wedge\ P_n[y/this, e/p]) \vee \\ (y = null\ \wedge\ Q_e \left[\begin{array}{l} \$ < NullPExc > /\$, \\ new(\$, NullPExc)/excV \end{array} \right]\ ) \\ \end{array} \right\}\ x = y.T : m(e)\ \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & Q_n[x/result]) \wedge \\ (\mathcal{X} = break & \Rightarrow & false) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e) \end{array} \right\}}$$

3.3 Methods

3.3.1 Implementation rule

$$\frac{\left\{\ P_n\ \right\}\ \ body(T@m)\ \left\{\begin{array}{lll}(\mathcal{X} = normal & \Rightarrow & Q_n)\ \wedge \\ (\mathcal{X} = break & \Rightarrow & false)\ \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e)\end{array}\right\}}{\left\{\ P_n\ \right\}\ \ T@m\ \left\{\begin{array}{lll}(\mathcal{X} = normal & \Rightarrow & Q_n)\ \wedge \\ (\mathcal{X} = break & \Rightarrow & false)\ \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e)\end{array}\right\}}$$

3.3.2 Class rule

$$\frac{\begin{array}{l} m\ is\ method\ in\ class\ T\ (i.e.\ impl(T,m)\ is\ defined) \\ \left\{\ \tau(this) = T\ \wedge\ P_n\ \right\}\ \ impl(T,m)\ \left\{\begin{array}{lll}(\mathcal{X} = normal & \Rightarrow & Q_n)\ \wedge \\ (\mathcal{X} = break & \Rightarrow & false)\ \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e)\end{array}\right\} \\ \left\{\ \tau(this) \prec T\ \wedge\ P_n\ \right\}\ \ T:m\ \left\{\begin{array}{lll}(\mathcal{X} = normal & \Rightarrow & Q_n)\ \wedge \\ (\mathcal{X} = break & \Rightarrow & false)\ \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e)\end{array}\right\}\end{array}}{\left\{\ P_n\ \right\}\ \ T:m\ \left\{\begin{array}{lll}(\mathcal{X} = normal & \Rightarrow & Q_n)\ \wedge \\ (\mathcal{X} = break & \Rightarrow & false)\ \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e)\end{array}\right\}}$$

3.3.3 Subtype rule

$$\frac{\begin{array}{l} S \preceq T \\ \left\{\ P_n\ \right\}\ \ S:m\ \left\{\begin{array}{lll}(\mathcal{X} = normal & \Rightarrow & Q_n)\ \wedge \\ (\mathcal{X} = break & \Rightarrow & false)\ \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e)\end{array}\right\}\end{array}}{\left\{\ \tau(this) \preceq S\ \wedge\ P_n\ \right\}\ \ T:m\ \left\{\begin{array}{lll}(\mathcal{X} = normal & \Rightarrow & Q_n)\ \wedge \\ (\mathcal{X} = break & \Rightarrow & false)\ \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e)\end{array}\right\}}$$

3.3.4 Block rule

$$\frac{\left\{\ P_n\ \land\ v_1 = init(T_1)\ \land\ ...\land\ v_n = init(T_n)\ \right\}\ s\ \left\{\begin{array}{ll}(\mathcal{X} = normal & \Rightarrow\ Q_n)\ \land\\ (\mathcal{X} = break & \Rightarrow\ false)\ \land\\ (\mathcal{X} = exc & \Rightarrow\ Q_e)\end{array}\right\}}{\left\{\ P_n\ \right\}\ T_1\ v1;\ ...\ T_n\ v_n;\ s\ \left\{\begin{array}{ll}(\mathcal{X} = normal & \Rightarrow\ Q_n)\ \land\\ (\mathcal{X} = break & \Rightarrow\ false)\ \land\\ (\mathcal{X} = exc & \Rightarrow\ Q_e)\end{array}\right\}}$$

3.4 While statement including breaks and exceptions

The execution of the statement s_1 can produce three possible results:

- either s_1 finishes in normal execution and I_n holds, or
- s_1 executes a break statement and Q_b holds, or
- s_1 throws an exception and R_e holds.

The while's postcondition is either the while finishes in a normal execution and ($(I_n \land \neg e) \lor Q_b$) holds or it finishes with an exception and R_e holds. After the while, we set $\mathcal{X} = break$ to false because if a break statement was executed in s_1 after the while we have normal execution.

$$\frac{\left\{\ e\ \land\ I_n\ \right\}\ s_1\ \left\{\begin{array}{ll}(\mathcal{X} = normal & \Rightarrow\ I_n)\ \land\\ (\mathcal{X} = break & \Rightarrow\ Q_b)\ \land\\ (\mathcal{X} = exc & \Rightarrow\ R_e)\end{array}\right\}}{\left\{\ I_n\ \right\}\ while\ (e)\ s_1\ \left\{\begin{array}{ll}(\mathcal{X} = normal & \Rightarrow\ (I_n \land \neg e) \lor\ Q_b)\ \land\\ (\mathcal{X} = break & \Rightarrow\ false)\ \land\\ (\mathcal{X} = exc & \Rightarrow\ R_e)\end{array}\right\}}$$

3.4.1 Break statement

In this rule, we change the value of $\mathcal{X}$ to *break*.

$$\frac{}{\left\{\ P_n\ \right\}\ break;\ \left\{\begin{array}{ll}(\mathcal{X} = normal & \Rightarrow\ false)\ \land\\ (\mathcal{X} = break & \Rightarrow\ P_n)\ \land\\ (\mathcal{X} = exc & \Rightarrow\ false)\end{array}\right\}}$$

3.5 Exception Handling

The logic for try catch statements has one rule covering all possible results of the statements s_1 and s_2. The execution of the statement s_1 can produce four different results:

1. s_1 finishes in normal execution and Q_n holds. In this case, the statement s_2 is not executed and the postcondition of the try-catch is the postcondition of s_1 ($\mathcal{X} = normal \Rightarrow Q_n$).

2. s_1 terminates with a break and Q_b holds. In this case, s_2 is again not executed and the postcondition of the try-catch is $\mathcal{X} = \mathit{break} \Rightarrow Q_b$.

3. s_1 raises an exception and the exception is not caught. The statement s_2 is not executed and the try-catch finishes in a exception mode. The postcondition is $\mathcal{X} = \mathit{exc} \Rightarrow (Q_e \wedge \tau(\mathit{excV}) \npreceq T)$

4. s_1 raises an exception and the exception is caught. In the postcondition of s_1, $(Q'_e \wedge \tau(\mathit{excV}) \preceq T)$ expresses that the exception is caught. Finally we execute the s_2 statement producing the postcondition. Note here, that the postcondition is not $\{\mathcal{X} = \mathit{normal} \;\Rightarrow\; R\}$. It is due to the s_2 statement can also throw an exception.

$$\dfrac{\begin{array}{l}\left\{\ P_n\ \right\}\ \ s_1\ \left\{\begin{array}{lll}(\mathcal{X} = \mathit{normal} & \Rightarrow & Q_n)\ \wedge\\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q_b)\ \wedge\\ (\mathcal{X} = \mathit{exc} & \Rightarrow & \left(\begin{array}{l}(Q_e\ \wedge\ \tau(\mathit{excV}) \npreceq T)\vee\\ (Q'_e\ \wedge\ \tau(\mathit{excV}) \preceq T)\end{array}\right))\end{array}\right\}\\ \left\{\ Q'_e[e/\mathit{excV}]\ \right\}\ \ s_2\ \left\{\begin{array}{lll}(\mathcal{X} = \mathit{normal} & \Rightarrow & Q_n)\ \wedge\\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q_b)\ \wedge\\ (\mathcal{X} = \mathit{exc} & \Rightarrow & R_e)\end{array}\right\}\end{array}}{\left\{\ P_n\ \right\}\ \ \mathit{try}\ s_1\ \mathit{catch}\ (T\ e)\ s_2\ \left\{\begin{array}{lll}(\mathcal{X} = \mathit{normal} & \Rightarrow & Q_n)\ \wedge\\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q_b)\ \wedge\\ (\mathcal{X} = \mathit{exc} & \Rightarrow & R_e\ \vee\ (Q_e\ \wedge\ \tau(\mathit{excV}) \npreceq T))\end{array}\right\}}$$

3.5.1 Throw Statement

The rule for *throw* statement modifies the postcondition P by updating the exception component of the state with the reference just evaluated.

$$\dfrac{}{\left\{\ P_n[e/\mathit{excV}]\ \right\}\ \ \mathit{throw}\ e\ \left\{\begin{array}{lll}(\mathcal{X} = \mathit{normal} & \Rightarrow & \mathit{false})\ \wedge\\ (\mathcal{X} = \mathit{break} & \Rightarrow & \mathit{false})\ \wedge\\ (\mathcal{X} = \mathit{exc} & \Rightarrow & P_n)\end{array}\right\}}$$

3.5.2 Finally Rule for Java

The logic for finally statements for Java is different from the logic for finally statements for C#. The C# compiler does not allow to write break statements inside of finally clauses. The C# compiler produces the Error CS0157 [12].

If an exception occurs in the body of the *try* clause, it will be re-raised after the statement s_2. If both the body of *try* clause and the body of *finally* clause throw an exception, the second one takes precedence. Let $eTmp$ be a fresh variable. We use $eTmp$ to store the exception occurred in s_1 because another might be raised and caught in s_2. After this, we still need to have access to the first exception of s_1 because this exception is the overall result of that statement [19].

Let $\mathcal{X}Tmp$ be another fresh variable. We use $\mathcal{X}Tmp$ to store the status of the program after the execution of s_1. Depending of the status after the execution of s_2 we need to propagate an exception or not or change the status of the program to break or not.

If an exception is raised in the statement s_1 then we have three possible cases:

1. the statement s_2 terminates normally, or

2. the statement s_2 terminates with a break, or

3. the statement s_2 raises another exception.

In the first case, the statement s_2 finishes in the postcondition Q'_e, but due to an exception was thrown in s_1, the status of the program is $\mathcal{X} = exc$.

The second case is may be, the strange one. s_2 executes a break statement then the status of the program is changed to $\mathcal{X} = break$ and the exception is cleaned. The program finishes in the postcondition $\mathcal{X} = break \Rightarrow Q'_b$.

In the last case, if s_2 thrown an exception, it takes precedence.

If there is not nether an exception nor a break statement in s_1 then the status of the program will be the produced by the statement s_2.

If a break statement is executed in s_1 then we have again three possible cases. If the statement s_2 finishes in normal execution, then the status of the program is $\mathcal{X} = break$ due to s_1 finished in break status. If s_2 executes another break, then the try-finally statement also finishes in break status. But if s_2 throws an exception, then the try-finally finishes in an exception status.

$$\left\{\ P_n\ \right\}\ s_1 \left\{ \begin{array}{ll} (\mathcal{X} = normal & \Rightarrow\ Q_n) \wedge \\ (\mathcal{X} = break & \Rightarrow\ Q_b) \wedge \\ (\mathcal{X} = exc & \Rightarrow\ Q_e) \end{array} \right\}$$

$$\frac{\left\{ \begin{array}{c} (Q_n\ \wedge\ \mathcal{X}Tmp = normal) \vee \\ (Q_b\ \wedge\ \mathcal{X}Tmp = break) \vee \\ \left(\begin{array}{l} (Q_e[eTmp/excV] \wedge \\ \mathcal{X}Tmp = exc \wedge \\ eTmp = excV) \end{array} \right) \end{array} \right\}\ s_2 \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & \left(\begin{array}{l} (Q'_n\ \wedge\ \mathcal{X}Tmp = normal) \vee \\ (Q'_b\ \wedge\ \mathcal{X}Tmp = break) \vee \\ (Q'_e\ \wedge\ \mathcal{X}Tmp = exc) \end{array} \right)) \wedge \\ (\mathcal{X} = break & \Rightarrow & Q'_b) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q'_e) \end{array} \right\}}{\left\{\ P_n\ \right\}\ try\ s_1\ finally\ s_2 \left\{ \begin{array}{ll} (\mathcal{X} = normal & \Rightarrow\ Q'_n) \wedge \\ (\mathcal{X} = break & \Rightarrow\ Q'_b) \wedge \\ (\mathcal{X} = exc & \Rightarrow\ Q'_e) \end{array} \right\}}$$

3.6 Language-independent rules

False axiom

$$\frac{}{\left\{\ \mathit{false}\ \right\}\ s_1 \left\{ \begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & \mathit{false}) \land \\ (\mathcal{X} = \mathit{break} & \Rightarrow & \mathit{false}) \land \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & \mathit{false}) \end{array} \right\}}$$

Strength

$$\frac{\begin{array}{l} P'_n \Rightarrow P_n \\ \left\{\ P_n\ \right\}\ s_1 \left\{ \begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q_n) \land \\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q_b) \land \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & Q_e) \end{array} \right\} \end{array}}{\left\{\ P'_n\ \right\}\ s_1 \left\{ \begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q_n) \land \\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q_b) \land \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & Q_e) \end{array} \right\}}$$

Weak

$$\frac{\begin{array}{l} \left\{\ P_n\ \right\}\ s_1 \left\{ \begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q_n) \land \\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q_b) \land \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & Q_e) \end{array} \right\} \\ Q_n \Rightarrow Q'_n \\ Q_b \Rightarrow Q'_b \\ Q_e \Rightarrow Q'_e \end{array}}{\left\{\ P_n\ \right\}\ s_1 \left\{ \begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q'_n) \land \\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q'_b) \land \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & Q'_e) \end{array} \right\}}$$

Invariant

$$\frac{\left\{\ P_n\ \right\}\ s_1 \left\{ \begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q_n) \land \\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q_b) \land \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & Q_e) \end{array} \right\}}{\left\{\ P_n \land W\ \right\}\ s_1 \left\{ \begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q_n \land W) \land \\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q_b \land W) \land \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & Q_e \land W) \end{array} \right\}}$$

Substitution

$$\frac{\left\{\ P_n\ \right\}\ s_1 \left\{ \begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q_n) \land \\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q_b) \land \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & Q_e) \end{array} \right\}}{\left\{\ P_n[t/Z]\ \right\}\ s_1 \left\{ \begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q_n[t/Z]) \land \\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q_b[t/Z]) \land \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & Q_e[t/Z]) \end{array} \right\}}$$

Conjunction

$$\frac{\begin{array}{l} \left\{\ P^1_n\ \right\}\ s_1 \left\{ \begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q^1_n) \land \\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q^1_b) \land \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & Q^1_e) \end{array} \right\} \\ \left\{\ P^2_n\ \right\}\ s_1 \left\{ \begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q^2_n) \land \\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q^2_b) \land \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & Q^2_e) \end{array} \right\} \end{array}}{\left\{\ P^1_n \land P^2_n\ \right\}\ s_1 \left\{ \begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q^1_n \land Q^2_n) \land \\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q^1_b \land Q^2_b) \land \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & Q^1_e \land Q^2_e) \end{array} \right\}}$$

Disjunction

$$\frac{\begin{array}{l} \left\{\ P^1_n\ \right\}\ s_1 \left\{ \begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q^1_n) \land \\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q^1_b) \land \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & Q^1_e) \end{array} \right\} \\ \left\{\ P^2_n\ \right\}\ s_1 \left\{ \begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q^2_n) \land \\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q^2_b) \land \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & Q^2_e) \end{array} \right\} \end{array}}{\left\{\ P^1_n \lor P^2_n\ \right\}\ s_1 \left\{ \begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q^1_n \lor Q^2_n) \land \\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q^1_b \lor Q^2_b) \land \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & Q^1_e \lor Q^2_e) \end{array} \right\}}$$

all-rule

$$\frac{\left\{\ P_n[Y/Z]\ \right\}\ s_1 \left\{ \begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q_n) \land \\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q_b) \land \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & Q_e) \end{array} \right\}}{\left\{\ P_n[Y/Z]\ \right\}\ s_1 \left\{ \begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & \forall Z : Q_n) \land \\ (\mathcal{X} = \mathit{break} & \Rightarrow & \forall Z : Q_b) \land \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & \forall Z : Q_e) \end{array} \right\}}$$

where Z, Y are arbitrary, but distinct logical variables.

ex-rule

$$\frac{\left\{\ P_n[Y/Z]\ \right\}\ s_1 \left\{ \begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & Q_n) \land \\ (\mathcal{X} = \mathit{break} & \Rightarrow & Q_b) \land \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & Q_e) \end{array} \right\}}{\left\{\ P_n[Y/Z]\ \right\}\ s_1 \left\{ \begin{array}{lll} (\mathcal{X} = \mathit{normal} & \Rightarrow & \exists Z : Q_n) \land \\ (\mathcal{X} = \mathit{break} & \Rightarrow & \exists Z : Q_b) \land \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & \exists Z : Q_e) \end{array} \right\}}$$

where Z, Y are arbitrary, but distinct logical variables.

3.7 Application of the COOL logic

In this subsection, we present three examples of the application of the logic for COOL. The first example (subsection 3.7.1) shows the application of while and break rules. The second one (subsection 3.7.2) shows the application of the logic for try catch and throw statements and also the invocation rule. The last one (subsection 3.7.3) shows the application of the logic for try finally statements inside of a loop body. Every example shows the source program and the proof.

3.7.1 Application of the logic for while and break statements

This example is a function that returns true if all elements i of the array a are equals to the summatory of the elements from 0 to $i-1$. We can write a predicate such as:

$$\forall i:\ 0 \le i < a.length:\ (\ a[i] = \sum_{j=0}^{i-1} a[j]\ )$$

```
/*@
@    public normal_behavior
@         requires a != null && a.length>0;
@         ensures (\result && (\forall int j; 0<=j && j<a.length;
@                                  a[j] = (\sum int k; 0<=k && k<j; a[k]) ) ) ||
@                   ( !\result && (\exists int j; 0<=j && j<a.length;
@                                  a[j] != (\sum int k; 0<=k && k<j; a[k]) ) )
@*/
public boolean example(int a []) {
    int ind = 1;
    int sum = a[0];
    boolean result=true;
    while (ind < a.length) {
        if (a[ind] != sum) {
            result = false;
            break;
        }
        sum = sum+a[ind];
        ind = ind+1;
    }
}
```

Figure 2: Source program for the application of while and break statement.

Table 1 shows the proof for the example figure 2.

Table 1: Proof for the example of figure 2.

```
public boolean example( int a [] ) {
```
$\{\ a \neq null\ \wedge\ a.length > 0\ \}$
```
    int ind=1;
```
$\{\ a \neq null\ \wedge\ a.length > 0\ \wedge\ ind = 1\ \}$
```
    int sum=a[0];
```
$\{\ ind = 1\ \wedge\ sum = a[0]\ \}$
```
    boolean result=true;
```
$\{\ ind = 1\ \wedge\ sum = a[0]\ \wedge\ result = true\ \}$

Continued on next page

```
    while (ind < a.length) {
        $\left\{\ \mathcal{X} = normal \ \Rightarrow\ (ind < a.length \ \wedge\ I)\ \right\}$
        if (a[ind] != sum) {
            $\left\{\ \mathcal{X} = normal \ \Rightarrow\ (ind < a.length \ \wedge\ I \ \wedge\ a[ind] \ \neq sum)\ \right\}$
            result=false;
            $\left\{\ \mathcal{X} = normal \ \Rightarrow\ (ind < a.length \ \wedge\ I \ \wedge\ a[ind] \ \neq sum \ \wedge\ result = false)\ \right\}$
            $\Rightarrow$
            $\left\{\ \mathcal{X} = normal \ \Rightarrow\ result = False \wedge\ \exists\, j:\ 0 \le j < a.length:\ (a[j] \neq \sum_{k=0}^{j-1}\ a[k])\ \right\}$
            break;
          $\left\{\ \mathcal{X} = break \ \Rightarrow\ (\ result = False \wedge\ \exists\, j:\ 0 \le j < a.length:\ (a[j] \neq \sum_{k=0}^{j-1}\ a[k])\ )\ \right\}$
        }
        $\left\{ \begin{array}{l} \mathcal{X} = normal \Rightarrow\ (\ ind < a.length \ \wedge\ I \ \wedge\ a[ind] = \sum_{k=0}^{ind-1} a[k]\ ) \wedge \\ \mathcal{X} = break \ \Rightarrow\ (\ result = False \wedge\ \exists\, j:\ 0 \le j < a.length:\ (a[j] \neq \sum_{k=0}^{j-1}\ a[k])\ ) \end{array} \right\}$
        sum=sum+a[ind];
        $\left\{ \begin{array}{l} \mathcal{X} = normal \Rightarrow \left( \begin{array}{l} ind < a.length \ \wedge\ a[ind] = \sum_{k=0}^{ind-1} a[k] \ \wedge \\ sum = (\sum_{k=0}^{ind-1} a[k]) + a[ind] \ \wedge \\ result \ \wedge\ (\forall\, j:\ 0 \le j < ind:\ (\ a[j] = \sum_{k=0}^{j-1} a[k]\ )) \wedge \\ 0 \le ind \le a.length \end{array} \right) \wedge \\ \mathcal{X} = break \ \Rightarrow\ (\ result = False \wedge\ \exists\, j:\ 0 \le j < a.length:\ (a[j] \neq \sum_{k=0}^{j-1}\ a[k])\ ) \end{array} \right\}$
       ind=ind+1;
        $\left\{ \begin{array}{l} \mathcal{X} = normal \Rightarrow\ I \ \wedge \\ \mathcal{X} = break \ \Rightarrow\ (\ result = False \wedge\ \exists\, j:\ 0 \le j < a.length:\ (a[j] \neq \sum_{k=0}^{j-1}\ a[k])\ ) \end{array} \right\}$
    }
    $\left\{\ \mathcal{X} = normal \Rightarrow \left( \begin{array}{l} (result \ \wedge\ \forall\, j:\ 0 \le j < a.length:\ (\ a[j] = \sum_{k=0}^{j-1} a[k]\ )) \vee \\ (result = False \wedge\ \exists\, j:\ 0 \le j < a.length: (a[j] \neq \sum_{k=0}^{j-1}\ a[k])) \end{array} \right)\ \right\}$
}
```

where

$$I \equiv \Big(res \ \wedge\ (\forall\, j:\ 0 \le j < ind:\ (\ a[j] = \sum_{k=0}^{j-1} a[k]\ )) \wedge\ sum = \sum_{k=0}^{ind-1} a[k] \ \wedge\ 0 \le ind \le a.length\Big)$$

3.7.2 Application of the logic for try catch and throw statements

This example shows the application of the logic for try catch statements. The method *exampleTry* invokes the method *myMet* inside of a try catch statement. If the method *myMet* throws an exception, then the result is $result = 1$ otherwise (in normal execution) $result = 0$. The method *myMet* terminates normally and returns an exception if the value of the variable b is $false$ otherwise it terminates abruptly and it throws an exception.

```
/*@
@    public  normal_behavior
@         requires     true;
@         ensures      (!b && \result=0) || (b && \result=1);
@*/
public int exampleTry (boolean b ) {
     int result;
     Exception auxExc;
     try {
          auxExc = myMet (b);
          result=0;
     }
     catch (Exception e) {
          result=1;
     }
}
/*@
@    public  normal_behavior
@         requires     !b;
@         ensures      \fresh (\ result ) && \typeof(\result)=\TYPE(Exception);
@ also
@    public  exceptional_behavior
@         requires     b;
@         signals_only  Exception;
@*/
public Exception myMet(boolean b) throws java.lang.Exception {
     if (b)
          throw new Exception();
     else
          result=new Exception();
}
```

Figure 3: Source program of the application of try-catch statement.

Table 2: Proof of the example figure 3 .

```
public int exampleTry (boolean b ) {
    int result;
    Exception auxExc;
    $\left\{ b = B \ \wedge \ OS = \$ \right\}$
    try {
        auxExc = myMet (b);
        $\left\{ \begin{array}{l} (\mathcal{X} = normal \ \Rightarrow \ (auxExc = new(\$, Exception) \ \wedge \ \$ = \$ < Exception > \ \wedge \neg B) \ ) \ \wedge \\ (\mathcal{X} = exc \Rightarrow \ B \ ) \end{array} \right\}$
        result =0;
        $\left\{ \begin{array}{l} (\mathcal{X} = normal \ \Rightarrow \ (auxExc = new(\$, Exception) \ \wedge \ \neg B \ \wedge \ result = 0) \ ) \ \wedge \\ (\mathcal{X} = exc \Rightarrow \ B \ ) \end{array} \right\}$
    }
    catch (Exception e) {
        $\left\{ \mathcal{X} = normal \Rightarrow \ B \right\}$
        result=1;
        $\left\{ \mathcal{X} = normal \Rightarrow \ (B \ \wedge \ result = 1 \right\}$
    }
```

Continued on next page

```
    $\left\{ \mathcal{X} = normal \Rightarrow \begin{pmatrix} (auxExc = new(\$, Exception) \wedge \neg B \wedge result = 0) \vee \\ (B \wedge result = 1) \end{pmatrix} \right\}$
}

public Exception myMet(boolean b) throws java.lang.Exception {
    $\left\{ b = B \wedge OS = \$ \right\}$
    if (b)
        $\left\{ \mathcal{X} = normal \Rightarrow (B \wedge OS = \$) \right\}$
        throw new Exception();
        $\left\{ \mathcal{X} = exc \Rightarrow B \right\}$
    else
        $\left\{ \mathcal{X} = normal \Rightarrow (\neg B \wedge OS = \$) \right\}$
        result = new Exception();
        $\left\{ \mathcal{X} = normal \Rightarrow (result = new(\$, Exception) \wedge \$ = \$ < Exception > \wedge \neg B) \right\}$

    $\left\{ \begin{array}{l} (\ \mathcal{X} = normal \Rightarrow (result = new(\$, Exception) \wedge \$ = \$ < Exception > \wedge \neg B)\ ) \wedge \\ (\ \mathcal{X} = exc \Rightarrow B\ ) \end{array} \right\}$
}
```

3.7.3 Application of the logic for try finally statements

This example shows the application of the logic for try finally statements. We assign $result = 5$ inside of the second try statement and then we execute a break statement if $a[i] > 0$. The two finally statements will be executed assigning $result++$ but the assignment $result = result + 20$ will not be executed in the same case. The result for this case is $result = 7$ and normal execution after the while statement. In the second case, we assign $result = 10$ and then $result = result + 20$, $result++$, $result++$ and $i++$. The postcondition is $result = 32$.

Table 3: Proof of the example figure 4 .

```
public int exampleTryFinally (int [ ] a ) {
    $\left\{ a \neq null \wedge 0 < a.length \right\}$
    int result=0;
    $\left\{ a \neq null \wedge 0 < a.length \wedge result = 0 \right\}$
    int i=0;
    $\left\{ a \neq null \wedge 0 < a.length \wedge result = 0 \wedge i = 0 \right\}$
    while (i< a.length ) {
        $\left\{ i < a.length \right\}$
        try {
            try {
                if (a[i]>0) {
                    $\left\{ i < a.length \wedge a[i] > 0 \right\}$
                    result=5;
                    $\left\{ i < a.length \wedge a[i] > 0 \wedge result = 5 \right\}$
```

Continued on next page

```
/*@
@     public   normal_behavior
@           requires    a != null && 0 < a. length ;
@           ensures     \ result =32 || \ result =7;
@*/
public int exampleTryFinally (int [ ] a ) {
    int result=0;
    int i=0;
    while (i< a.length ) {
        try {
            try {
                if (a[i]>0) {
                    result=5;
                    break;
                }
                result=10;
            }
            finally {
                result=result+1;
            }
            result=result+20;
        }
        finally {
            result=result+1;
        }
        i=i+1;
    }
}
```

Figure 4: Source program of the application of try-finally statement.

```
                break;
```
$$\left\{ \mathcal{X} = break \;\Rightarrow\; (i < a.length \;\wedge\; a[i] > 0 \;\wedge\; result = 5) \right\}$$
```
            }
            result=10;
```
$$\left\{ \begin{array}{l} \mathcal{X} = normal \;\Rightarrow\; (i < a.length \;\wedge\; result = 10) \\ \mathcal{X} = break \;\Rightarrow\; (i < a.length \;\wedge\; a[i] > 0 \;\wedge\; result = 5) \end{array} \right\}$$
```
        }
        finally {
```
$$\left\{ \begin{array}{l} (\mathcal{X}Tmp = normal \;\wedge\; i < a.length \;\wedge\; result = 10) \vee \\ (\mathcal{X}Tmp = break \;\wedge\; i < a.length \;\wedge\; a[i] > 0 \;\wedge\; result = 5) \end{array} \right\}$$
```
            result=result+1;
```
$$\left\{ \begin{array}{l} (\mathcal{X}Tmp = normal \;\wedge\; i < a.length \;\wedge\; result = 11) \vee \\ (\mathcal{X}Tmp = break \;\wedge\; i < a.length \;\wedge\; a[i] > 0 \;\wedge\; result = 6) \end{array} \right\}$$
```
        }
```
$$\left\{ \begin{array}{l} \mathcal{X} = normal \;\Rightarrow\; (i < a.length \;\wedge\; result = 11) \\ \mathcal{X} = break \;\Rightarrow\; (i < a.length \;\wedge\; a[i] > 0 \;\wedge\; result = 6) \end{array} \right\}$$
```
        result=result+20;
```

Continued on next page

```
            $\left\{ \begin{array}{l} \mathcal{X} = \mathit{normal} \;\Rightarrow\; (i < a.length \;\wedge\; result = 31) \\ \mathcal{X} = \mathit{break} \;\Rightarrow\; (i < a.length \;\wedge\; a[i] > 0 \;\wedge\; result = 6) \end{array} \right\}$
        }
        finally {
            $\left\{ \begin{array}{l} (\mathcal{X}Tmp = \mathit{normal} \;\wedge\; i < a.length \;\wedge\; result = 31) \vee \\ (\mathcal{X}Tmp = \mathit{break} \;\wedge\; i < a.length \;\wedge\; a[i] > 0 \;\wedge\; result = 6) \end{array} \right\}$
            result=result+1;
            $\left\{ \begin{array}{l} (\mathcal{X}Tmp = \mathit{normal} \;\wedge\; i < a.length \;\wedge\; result = 32) \vee \\ (\mathcal{X}Tmp = \mathit{break} \;\wedge\; i < a.length \;\wedge\; a[i] > 0 \;\wedge\; result = 7) \end{array} \right\}$
        }
        $\left\{ \begin{array}{l} \mathcal{X} = \mathit{normal} \;\Rightarrow\; (i < a.length \;\wedge\; result = 32) \\ \mathcal{X} = \mathit{break} \;\Rightarrow\; (i < a.length \;\wedge\; a[i] > 0 \;\wedge\; result = 7) \end{array} \right\}$
        i=i+1;
        $\left\{ \begin{array}{l} \mathcal{X} = \mathit{normal} \;\Rightarrow\; (i \le a.length \;\wedge\; result = 32) \\ \mathcal{X} = \mathit{break} \;\Rightarrow\; (i < a.length \;\wedge\; a[i] > 0 \;\wedge\; result = 7) \end{array} \right\}$
    }
    $\left\{ \mathcal{X} = \mathit{normal} \;\Rightarrow\; \left( \begin{array}{l} (i = a.length \;\wedge\; result = 32) \vee \\ (i < a.length \;\wedge\; a[i] > 0 \;\wedge\; result = 7) \end{array} \right) \right\}$
    $\Rightarrow$
    $\left\{ \mathcal{X} = \mathit{normal} \;\Rightarrow\; (result = 32 \vee result = 7) \right\}$
}
```

4 The Bytecode Language

The bytecode language consists of classes with fields and methods. Methods are implemented as method bodies consisting of a sequence of labeled bytecode instructions. Bytecode instructions operate on an evaluation stack (sometimes called operand stack), local variables (which also include parameters), and the object store (heap). The following list gives an informal overview of the instructions available in the bytecode language, the operational semantics is presented in [1, 2].

- pushc v: pushes a constant v onto the stack.
- pushv x: pushes the value of a local variable (or method parameter) x onto the stack.
- pop x: pops the top element off the stack and assigns it to the local variable x.
- op_{op} : assuming that op is a function that takes n input values to m output values, it removes the n top elements from the stack by applying op to them and puts the m output values onto the stack. We write bin_{op} if op is a binary function.
- goto l: transfers the control flow to the point l.
- brtrue l: transfers the control flow to the point l if the top element of the stack is true and unconditionally pops it.
- brfalse l: transfers the control flow to the point l if the top element of the stack is false and unconditionally pops it.
- checkcast T: checks whether the top element is of type T or a subtype thereof.
- newobj T: allocates a new object of type T and pushes it onto the stack.

- invokevirtual M and call M : invokes the method M on an optional object reference and parameters on the stack and replaces these values by the return value of the invoked method (if M returns a value). call invokes non-virtual and static methods, invokevirtual invokes virtual methods. The code depends on the actual type of the object reference (dynamic dispatch).
- getfield F: replaces the top element by its field F.
- putfield F: sets the field F of the object denoted by the second-topmost element to the top element of the stack and pops both values.
- nop: has no effect.
- athrow: takes one argument from the stack (the exception type) and thrown an exception.
- iaload: pops index and arrayref of an integer array and pushes arrayref[index].
- arraylength: pops objectref of an array and pushes its length of that array.

5 The Bytecode Logic

The Hoare-style program logic presented in this section allows one to formally verify that implementations satisfy interface specifications given as pre- and postconditions. For more detail of the Bytecode logic see [1].

5.1 Method and Instructions Specifications

A *method implementation T@m* represents the concrete implementation of method m in class T. A *virtual method T:m* represents the common properties of all method implementations that might by invoked dynamically when m is called on a receiver of static type T, that is, *impl(T,m)* (if *T:m* is not abstract) and all overriding subclass methods.

Properties of methods and method bodies are expressed by Hoare triples of the form {P} *comp* {Q}, where P, Q are sorted first-order formulas and *comp* is a method implementation T@m, a virtual method T:m or a method body p. We call such a triple *method specification.* 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 in which Q holds, or (2) *comp* aborts due errors or actions that are beyond the semantics of the programming language (for instance, memory allocation problems), 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 instructions I_l in a method body p has a precondition E_l. An instruction with its precondition is called an instruction specification, written as $\{E_l\}\ l : I_l$.

Obviously, the meaning of an instruction specification $\{E_l\}\ l : I_l$ cannot be defined in isolation. $\{E_l\}\ l : I_l$ express 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 [1].

5.2 Rules for Instruction Specifications

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

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

$wp^1_p(I_l)$ is the *local weakest precondition* of instruction I_l. Such a rule express that the precondition of I_l has to imply the weakest precondition of I_l w.r.t. all possible successor instructions of I_l.

The definition of wp_p^1 is shown in figure 5. Within an assertion, the current stack is referred to as s, and its elements are denoted by non-negative integers: element 0 is the top element, etc. The interpretation $[E_l]$: State x Stack $\rightarrow$ Value for s is

$$[s(0)] < S, (\sigma, v) > = v \quad and$$

$$[s(i+1)] < S, (\sigma, v) >= [s(i)] < S, \sigma >$$

The functions $shift$ and $unshift$ express the substitutions that occur when values are pushed onto and popped from the stack, resp.:

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

$shift^n$ denotes n consecutive applications of $shift$.

I_l	$wp_p^1(I_l)$
pushc v	$unshift(E_{l+1}[v/s(0)])$
pushv x	$unshift(E_{l+1}[x/s(0)])$
pop x	$(shift(E_{l+1}))[s(0)/x]$
bin_{op}	$(shift(E_{l+1}))[s(1) op s(0)/s(1)]$
goto l'	$E_{l'}$
brtrue l'	$(\neg s(0) \Rightarrow shift(E_{l+1})) \wedge (s(0) \Rightarrow shift(E_{l'}))$
checkcast T	$E_{l+1} \wedge \tau(s(0)) \preceq T$
newobj T	$unshift(E_{l+1}[new(\$, T)/s(0), \$ < T > /\$]$
getfield $T@a$	$E_{l+1}[\$(iv(s(0), T@a))/s(0)] \;\wedge\; s(0) \neq null$
putfield $T@a$	$(shift^2(E_{l+1}))[\$ < iv(s(1), T@a) := s(0) > /\$] \;\wedge\; s(1) \neq null$
return	true
areturn	(shift(Q))[s(0)/result] where Q is the method's postcondition.
nop	E_{l+1}

Figure 5: The values of the wp_p^1 function.

6 Proof transformation from O-O Programs to Bytecode

A proof-transforming compiler is based on transformation functions, ∇_S and ∇_E, for statements and expressions, respectively. Both functions yield a sequence of Bytecode instructions and their specification. ∇_S generates this sequence from a proof for a source statement and ∇_E generates this from a source expression and a precondition for its evaluation. These functions are defined as a composition of the translations of its sub-trees. The signatures are the following:

$$\begin{array}{ll} \nabla_E & : Precondition \times Expression \times Postcondition \times Label \rightarrow BytecodeProof \\ \nabla_S & : ProofTree \times Map[Event \rightarrow Label] \times List[Finally] \times ExceptionTable \rightarrow \\ & \qquad [BytecodeProof \times ExceptionTable] \end{array}$$

In ∇_E the label is used to the starting label of the translation.

ProofTree is a proof tree to translate. It is a derivation in the Hoare logic. For example

$$\dfrac{\dfrac{Tree_1}{\{P\}\ s_1\ \{Q\}} \qquad \dfrac{Tree_2}{\{Q\}\ s_2\ \{R\}}}{\{P\}\quad s_1; s_2\quad \{R\}}$$

is a proof tree for the compositional rule where P, Q and R are preconditions and postconditions (predicates in first order logic) and s_1, s_2 statements.

Map is a mapping function: Event $\rightarrow$ Label. The *Event* type is defined as:

$$Event \quad := \quad start \quad | \quad next \quad | \quad break$$

The meaning is the following:

- start $\rightarrow l_s$: It is used to know the starting label of the translation.
- next $\rightarrow l_e$: It is used to know next label of the translation. For example, it is used in the translation of if else statements to know where to jump when the else translation finishes (after the then part).
- break $\rightarrow l_{break}$: It is used to process a unlabelled break statement. It means that we will transfer the control flow to the label l_{break} when we process a break. For unlabelled breaks, l_{break} represents the end of the loop we are processing.

In the mapping function *Event* $\rightarrow$ *Label*, the break labels are needed because in the beginning of a loop we know the exact point to transfer the control flow (the end of the loop). But after that, for example when we process a *break* instruction, we do not know where to transfer the control flow (if we would not have these labels).

The type *Finally* is defined as a tuple of [*ProofTree* , *ExceptionTable*]. It is used only to translate finally statements. When the javac compiler translates a break statement inside a try block with finally statements, it duplicates the finally code before the break statement and then it adds the break translation. We want to generate the same code as the javac compiler. So we use the list of *ProofTree* to store the proof tree of each finally clauses and the *ExceptionTable* to store the exception table of the finally block. When a break statement inside a try-finally statement is found, we duplicate the finally code. Note that we have a list of *Finally* because we could have several try-finally statements and then a break statement (see subsection 3.7.3 on page 19 for an example).

The *ExceptionTable* type is defined as:

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

6.1 Notation

The translation functions are defined over the Hoare rules of the source language. Each translation was derived by using the source proof and the wp_p^1 definition (figure 5). To clarify the notation, we use symbolic labels denoted by l_a, l_b, etc; for pre and post conditions we use capital letters P, Q, R, etc; for expressions e, e_1, etc; and for statements s_1, s_2, etc.

For example, when we translate

Let $[B_{S_1}, et_1]$ and $[B_{S_2}, et_2]$ be

$$[B_{S_1}, et_1] \quad = \nabla_S \left(\frac{Tree_1}{\{P\}\ s_1\ \{Q\}},\ m\left[\ \text{next} \ \rightarrow l_b\ \right],\ f,\ et \right)$$

$$[B_{S_2}, et_2] \quad = \nabla_S \left(\frac{Tree_2}{\{Q\}\ s_2\ \{R\}},\ m\left[\ \text{start} \ \rightarrow l_b\ \right],\ f,\ et_1 \right)$$

$$\nabla_S\left(\frac{\dfrac{Tree_1}{\{P\}\ s_1\ \{Q\}} \qquad \dfrac{Tree_2}{\{Q\}\ s_2\ \{R\}}}{\{P\}\quad s_1; s_2\quad \{R\}}, m,\ f,\ et\right) = [B_{S_1} + B_{S_2}, et_2]$$

It means that the next label of the last instruction of the first ∇_S is l_b and there are not instructions between them.

$$m\left[\ \mathsf{start}\quad \rightarrow l_b\ \right]$$

means that we update the function m. start $\rightarrow\ l_b$ means that we replace the value of the parameter start by the label l_b.

We use symbolic labels denoted by l_a, l_b, l_c etc where $l_a = m[start]$; $l_b = succ(l_a)$; $l_c = succ(l_b)$ and $succ$ is a function for the generation of new labels ($succ : Label \rightarrow Label$).

6.1.1 Eliminating the variable $\mathcal{X}$ in the bytecode proof

The variable $\mathcal{X}$ is used to describe normal and abrupt termination (either exceptions or break statements) of the source program. The bytecode language has instructions to transfer the control flow (for example brtrue or goto). The bytecode logic does not need an special variable to express the status of the program because for example either break statements or continue statements are translated to unconditional jumps. In the case of exceptions (in the source logic), the variable $\mathcal{X}$ is used both to express that an exception has been thrown and to simulate that the control flow is transferred to either the catch block or the end of the method. On the bytecode logic, the only information we need is whether an exception was thrown or not and the type of it. This information is stored in the variable $excV$.

6.2 Starting the translation

Our Proof-Transforming Compiler takes as input a list of classes with their proof. Each class consists of a list of methods and its attributes. Each method consists of a proof tree. The PTC takes the first class and for every method of the class it invokes the translation function ∇_S setting the list of proof tree f to $\emptyset$ and the initial mapping function m as in the following example:

```
public class C {
    public C {
        Proof Tree of method C (PT_C)
    }
    public int m1 (int i) {
        Proof Tree of method m1 (PT_m1)
    }
    private String m2() {
        Proof Tree of method m2 (PT_m2)
    }
}
```

public class C {
  public C {
    Code:

$$\nabla_S \left(PT_C,\ m \begin{bmatrix} \textsf{start} & \rightarrow l_a \\ \textsf{next} & \rightarrow l_b \\ \textsf{break} & \rightarrow \emptyset \end{bmatrix},\ \emptyset,\ \emptyset \right)$$

l_b : return
  }
  public int m1 (**int** i) {
    Code:

$$\nabla_S \left(PT_{m1},\ m \begin{bmatrix} \textsf{start} & \rightarrow l_a \\ \textsf{next} & \rightarrow l_b \\ \textsf{break} & \rightarrow \emptyset \end{bmatrix},\ \emptyset,\ \emptyset \right)$$

l_b : return $result$
  }
  private String m2() {
    Code:

$$\nabla_S \left(PT_{m2},\ m \begin{bmatrix} \textsf{start} & \rightarrow l_a \\ \textsf{next} & \rightarrow l_b \\ \textsf{break} & \rightarrow \emptyset \end{bmatrix},\ \emptyset,\ \emptyset \right)$$

l_b : return $result$
  }
}
where l_a and l_b are fresh labels.

In the following we present the translation rules. In subsection 6.3 we present the expression translation. In subsection 6.4, we present the translation of language-independent rules for the source logic. In subsection 6.5 we present the translation for *assign*, *conditional* and *composition* statements. The references, objects and method invocation translation are presented in subsections 6.6 and 6.7, resp. After that, we include the treatment of *break* and *while* statements in subsection 6.8. Finally, we present the translation for exception handling in subsection 6.9.

6.3 Expression Translation

In this section we present the definition of ∇_E, the translation function for expressions. We consider constants, variables, unary and binary expressions.

6.3.1 Constants

$$\nabla_E(\ Q \wedge unshift(P[c/s(0)])\ ,\ \ c\ \ ,\ shift(Q) \wedge P\ ,\ \ l_a) =$$

$$\{\text{Q} \wedge \text{unshift(P}[c/s(0)])\} \quad l_a : \textsf{pushc c}$$

6.3.2 Variables

$$\nabla_E(\ Q \wedge unshift(P[x/s(0)])\ ,\ \ x\ \ ,\ shift(Q) \wedge P\ ,\ \ l_a) =$$

$$\{\text{Q} \wedge \text{unshift(P}[x/s(0)])\} \quad l_a : \textsf{pushv x}$$

6.3.3 Expressions: e_1 op e_2

$$\nabla_E(\ Q \wedge\ unshift(P[e_1\ op\ e_2/s(0)])\ ,\ \ e_1\ op\ e_2\ \ ,\ shift(Q) \wedge P\ ,\ \ l_a) =$$

$$\begin{array}{l}
\nabla_E(\ Q \ \wedge\ unshift(P[e_1 ope_2/s(0)])\ ,\ \ e_1\ \ ,\ shift(Q)\ \wedge\ P[s(0)\ op\ e_2/s(0)]\ ,\ l_a) \\
\nabla_E(\ shift(Q)\ \wedge\ P[s(0)\ op\ e_2/s(0)]\ ,\ \ e_2\ \ ,\ shift^2(Q)\ \wedge\ shift\ P[s(1)\ op\ s(0)/s(1)]\ ,\ l_b) \\
\ \{\ shift^2(Q)\ \wedge\ shift(P[s(1)\ op\ s(0)/s(1)])\ \}\quad l_c : binop_{op}
\end{array}$$

6.3.4 Expressions: unop e_2

$$\nabla_E(\ Q \wedge\ unshift(P[unop\ e/s(0)])\ ,\ \ unop\ e\ \ ,\ shift(Q)\wedge P\ ,\ l_a) =$$

$$\begin{array}{l}
\nabla_E(\ Q\ \wedge\ unshift(P[unop\ e/s(0)])\ ,\ \ e\ ,\ shift(Q)\ \wedge\ P[unop\ s(0)/s(0)]\ ,\ l_a) \\
\ \{shift(Q)\ \wedge\ P[unop\ s(0)/s(0)]\}\quad l_b : unop_{op}
\end{array}$$

6.4 Translation of language-independent Rules

In this section we summarize the translation of language-independent rules to Bytecode.

6.4.1 Strength

In the strength transformation we need translate $P'_n \Rightarrow P_n$ and $\{P_n\}\ s_1\ \{Q\}$. $P'_n \Rightarrow P_n$ can be translated by using the nop instruction. To translate $\{P_n\}\ s_1\ \{Q\}$ we use the ∇_S translation function.

Let b_{nop} and B_{S_1} be

$$b_{nop} = \ \{P'_n\}\ \ l_a : \mathsf{nop}$$

$$[B_{S_1}, et_1] = \ \nabla_S\left(\frac{Tree_1}{\{\ P_n\ \}\ \ s_1\ \left\{\begin{array}{lll}(\mathcal{X}=normal & \Rightarrow & Q_n)\ \wedge \\ (\mathcal{X}=break & \Rightarrow & Q_b)\ \wedge \\ (\mathcal{X}=exc & \Rightarrow & Q_e)\end{array}\right\}},\ m\left[\ \mathsf{start}\ \ \rightarrow l_b\ \right],\ f,\ et \right)$$

$$\nabla_S\left(\frac{\dfrac{Tree_1}{\{\ P_n\ \}\ \ s_1\ \left\{\begin{array}{lll}(\mathcal{X}=normal & \Rightarrow & Q_n)\ \wedge \\ (\mathcal{X}=break & \Rightarrow & Q_b)\ \wedge \\ (\mathcal{X}=exc & \Rightarrow & Q_e)\end{array}\right\}}\qquad P'_n \Rightarrow P_n}{\{\ P'_n\ \}\ \ s_1\ \left\{\begin{array}{lll}(\mathcal{X}=normal & \Rightarrow & Q_n)\ \wedge \\ (\mathcal{X}=break & \Rightarrow & Q_b)\ \wedge \\ (\mathcal{X}=exc & \Rightarrow & Q_e)\end{array}\right\}}\ ,\ m,\ f,\ et \right) =$$

$$[\ b_{nop} + B_{S_1}\ ,\ et_1\ ]$$

6.4.2 Weak

Similar to strength rule, in weak rule we translate $Q_n \Rightarrow Q'_n$ by using `nop`
Let B_{S_1} and b_{nop} be

$$\begin{aligned}
[B_{S_1}, et_1] = & \quad \nabla_S \left(\frac{Tree_1}{\left\{\ P_n\ \right\}\ \ s_1\ \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & Q_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & Q_b) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e) \end{array} \right\}},\ m,\ f,\ et \right) \\
b_{nop} = & \quad \{Q_n\}\ \ l_b : \mathsf{nop}
\end{aligned}$$

$$\nabla_S \left(\frac{\dfrac{Tree_1}{\left\{\ P_n\ \right\}\ \ s_1\ \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & Q_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & Q_b) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e) \end{array} \right\}} \quad \begin{array}{c} Q_n \Rightarrow Q'_n \\ Q_b \Rightarrow Q'_b \\ Q_e \Rightarrow Q'_e \end{array}}{\left\{\ P_n\ \right\}\ \ s_1\ \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & Q'_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & Q'_b) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q'_e) \end{array} \right\}},\ m,\ f,\ et \right) =$$

$$[\ B_{S_1} + b_{nop}\ ,\ et_1\ ]$$

6.4.3 Invariant

$$\nabla_S \left(\frac{\dfrac{Tree_1}{\left\{\ P_n\ \right\}\ \ s_1\ \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & Q_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & Q_b) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e) \end{array} \right\}}}{\left\{\ P_n\ \wedge\ W\ \right\}\ \ s_1\ \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & Q_n\ \wedge\ W) \wedge \\ (\mathcal{X} = break & \Rightarrow & Q_b\ \wedge\ W) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e\ \wedge\ W) \end{array} \right\}},\ m,\ f,\ et \right) =$$

We just add a conjunct W to every specification of the sequence produced by:

$$\nabla_S \left(\frac{Tree_1}{\left\{\ P_n\ \right\}\ \ s_1\ \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & Q_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & Q_b) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e) \end{array} \right\}},\ m,\ f,\ et \right)$$

6.4.4 Substitution

$$\nabla_S\left(\frac{\dfrac{Tree_1}{\{\ P_n\ \}\quad s_1\quad \left\{\begin{array}{lll}(\mathcal{X}=normal & \Rightarrow & Q_n)\ \wedge\\ (\mathcal{X}=break & \Rightarrow & Q_b)\ \wedge\\ (\mathcal{X}=exc & \Rightarrow & Q_e)\end{array}\right\}}}{\{\ P_n[t/Z]\ \}\quad s_1\quad \left\{\begin{array}{lll}(\mathcal{X}=normal & \Rightarrow & Q_n[t/Z])\ \wedge\\ (\mathcal{X}=break & \Rightarrow & Q_b[t/Z])\ \wedge\\ (\mathcal{X}=exc & \Rightarrow & Q_e[t/Z])\end{array}\right\}},\ m,\ f,\ et\right)=$$

As before, first we generate

$$\nabla_S\left(\frac{Tree_1}{\{\ P_n\ \}\quad s_1\quad \left\{\begin{array}{lll}(\mathcal{X}=normal & \Rightarrow & Q_n)\ \wedge\\ (\mathcal{X}=break & \Rightarrow & Q_b)\ \wedge\\ (\mathcal{X}=exc & \Rightarrow & Q_e)\end{array}\right\}},\ m,\ f,\ et\right)$$

and then we replace Z by t in each specification and in all proofs for assertions.

6.4.5 Conjunction/disjunction

Conjunction and disjunction are treated identically, so we present only the conjunction rule.

Let T_a be

$$\frac{Tree_1}{\{\ P^1\ \}\quad s_1\quad \left\{\begin{array}{lll}(\mathcal{X}=normal & \Rightarrow & Q_n^1)\ \wedge\\ (\mathcal{X}=break & \Rightarrow & Q_b^1)\ \wedge\\ (\mathcal{X}=exc & \Rightarrow & Q_e^1)\end{array}\right\}}$$

and let T_b be

$$\frac{Tree_2}{\{\ P^2\ \}\quad s_1\quad \left\{\begin{array}{lll}(\mathcal{X}=normal & \Rightarrow & Q_n^2)\ \wedge\\ (\mathcal{X}=break & \Rightarrow & Q_b^2)\ \wedge\\ (\mathcal{X}=exc & \Rightarrow & Q_e^2)\end{array}\right\}}$$

$$\nabla_S\left(\frac{T_a\qquad T_b}{\left\{\ P^1\ \wedge\ P^2\ \right\}\quad s_1\quad \left\{\begin{array}{lll}(\mathcal{X}=normal & \Rightarrow & Q_n^1\ \wedge\ Q_n^2)\ \wedge\\ (\mathcal{X}=break & \Rightarrow & Q_b^1\ \wedge\ Q_b^2)\ \wedge\\ (\mathcal{X}=exc & \Rightarrow & Q_e^1\ \wedge\ Q_e^2)\end{array}\right\}},\ m,\ f,\ et\right)=$$

$$\nabla_S\left(\frac{\{P_1\}\ s_1\ \{Q_1\}\quad \{P_2\}\ s_1\ \{Q_2\}}{\{P_1\ \wedge\ P_2\}\ s_1\ \{Q_1\ \wedge\ Q_2\}}\ ,\ m,\ f,\ et\right)=$$

We create the two proofs

$$\nabla_S\,(\ T_a,\ m,\ f,\ et)$$

and

$$\nabla_S\left(\ T_b,\ m\left[\ \mathsf{start}\ \ \rightarrow l_b\ \right],\ f,\ et\right)$$

The embedded instructions are by construction the same. With the two proofs, we assemble a third proof as result by merging, for all instructions, their specification from:

$$(\ \{A_{(l)}\}\ instr$$

and

$$(\ \{B_{(l)}\}\ instr$$

we obtain

$$\{A_{(l)}\ \wedge\ B_{(l)}\}\ instr$$

6.4.6 nops generated during the translation

The translation of language-independent rules produces nop instructions. javac compiler does not generated nop instructions and we wish to generate the same code as javac. nop instructions can be removed in a second pass though the bytecode proof. We replace nops instruction using the knowledge of the implication. For example, we can replace the following bytecode proof:

$$\begin{array}{ll} \{\ 0 \leq \mathrm{i} < \mathrm{n}\ \} & 00 : \mathsf{nop} \\ \{\ (0 \leq \mathrm{i} < \mathrm{n}) \wedge \mathrm{y{=}y}\} & 01 : \mathsf{pushv}\ \mathrm{i} \end{array}$$

by the following bytecode proof without nop instructions:

$$\{\ 0 \leq \mathrm{i} < \mathrm{n}\ \}\quad 00 : \mathsf{pushv}\ \mathrm{i}$$

6.5 Translation of Assign, conditional and compositional statements

6.5.1 Assign Statement

Let b_1 and b_2 be

$$\begin{array}{ll} B_e = & \nabla_E\ (\ P_n[e/x]\ ,\ \ e\ \ ,\ (shift(P_n[e/x])\ \wedge\ s(0) = e)\ ,\ m[start]) \\ b_{pop} = & \left\{\ shift(P_n[e/x]) \wedge\ s(0) = e\ \right\}\ \ l_b : \mathsf{pop}\ x \end{array}$$

$$\nabla_S\left(\frac{}{\left\{\ P_n[e/x]\ \right\}\quad x = e\quad \left\{\begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & P_n)\ \wedge \\ (\mathcal{X} = break & \Rightarrow & false)\ \wedge \\ (\mathcal{X} = exc & \Rightarrow & false) \end{array}\right\}},\ m,\ f,\ et\right) = [\ B_e + b_{pop}\ ,\ et\ ]$$

6.5.2 Conditional Statement

In this translation, we first obtain the translation for the expression e by using ∇_E. If e is true (e is on the top of the stack), we jump to l_e and we obtain the translation for the subtree s_1 by using ∇_S. Otherwise, we translate s_2 and then jump to the end (l_f).

Let T_{S_1} be the following proof tree:

$$T_{S_1} \equiv \cfrac{Tree_1}{\left\{\ P_n \ \wedge\ e \ \right\}\quad s_1 \quad \left\{\begin{array}{ll} (\mathcal{X} = normal & \Rightarrow\ Q_n)\ \wedge \\ (\mathcal{X} = break & \Rightarrow\ Q_b)\ \wedge \\ (\mathcal{X} = exc & \Rightarrow\ Q_e) \end{array}\right\}}$$

and T_{S_2} be

$$T_{S_2} \equiv \cfrac{Tree_2}{\left\{\ P_n \ \wedge\ \neg e \ \right\}\quad s_2 \quad \left\{\begin{array}{ll} (\mathcal{X} = normal & \Rightarrow\ Q_n)\ \wedge \\ (\mathcal{X} = break & \Rightarrow\ Q_b)\ \wedge \\ (\mathcal{X} = exc & \Rightarrow\ Q_e) \end{array}\right\}}$$

Let:

$$\begin{array}{rl} B_e = & \nabla_E\ (\ P_n,\ \ e\ \ ,\ (shift(P_n)\ \wedge\ s(0) = e)\ ,\ m[start]) \\ b_{brtrue} = & \{shift(P_n) \wedge s(0) = e\} \quad l_b : \mathsf{brtrue}\ l_e \end{array}$$

$$[B_{S_2}, et_1] = \quad \nabla_S\ \left(T_{S_2},\ m \begin{bmatrix} \mathsf{start} & \rightarrow l_c \\ \mathsf{next} & \rightarrow l_d \end{bmatrix},\ f,\ et\right)$$

$$\begin{array}{rl} b_{goto} = & \{Q_n\} \qquad\qquad\qquad l_d : \mathsf{goto}\ m[next] \\ [B_{S_1}, et_2] = & \nabla_S\ \left(T_{S_1},\ m \begin{bmatrix} \mathsf{start} & \rightarrow l_e \end{bmatrix},\ f,\ et_1\right) \end{array}$$

$$\nabla_S \left(\cfrac{T_{S_1} \qquad T_{S_2}}{\left\{\ P_n\ \right\}\ \ if\ (e)\ then\ s_1\ else\ s_2 \ \left\{\begin{array}{ll} (\mathcal{X} = normal & \Rightarrow\ Q_n)\ \wedge \\ (\mathcal{X} = break & \Rightarrow\ Q_b)\ \wedge \\ (\mathcal{X} = exc & \Rightarrow\ Q_e) \end{array}\right\}},\ m,\ f,\ et \right) =$$

$$[\ B_e + b_{brtrue} + B_{S_2} + b_{goto} + B_{S_1}\ ,\ et_2\ ]$$

6.5.3 Compositional Statement

Let T_{S_1} be the following proof tree:

$$T_{S_1} \equiv \cfrac{Tree_1}{\left\{\ P_n\ \right\}\quad s_1 \quad \left\{\begin{array}{ll} (\mathcal{X} = normal & \Rightarrow\ Q_n)\ \wedge \\ (\mathcal{X} = break & \Rightarrow\ R_b)\ \wedge \\ (\mathcal{X} = exc & \Rightarrow\ R_e) \end{array}\right\}}$$

and T_{S_2} be

$$T_{S_2} \equiv \frac{Tree_2}{\left\{ \; Q_n \; \right\} \quad s_2 \quad \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & R_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & R_b) \wedge \\ (\mathcal{X} = exc & \Rightarrow & R_e) \end{array} \right\}}$$

Let $[B_{S_1}, et_1]$ and $[B_{S_2}, et_2]$ be:

$$[B_{S_1}, et_1] = \nabla_S \Big(T_{S_1}, \; m \Big[\; \textsf{next} \; \rightarrow l_b \; \Big], \; f, et \Big)$$

$$[B_{S_2}, et_2] = \nabla_S \Big(T_{S_2}, \; m \Big[\; \textsf{start} \rightarrow l_b \; \Big], f, et_1 \Big)$$

$$\nabla_S \left(\frac{T_{S_1} \qquad\qquad T_{S_2}}{\left\{ \; P \; \right\} \quad s_1; s_2 \quad \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & R_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & R_b) \wedge \\ (\mathcal{X} = exc & \Rightarrow & R_e) \end{array} \right\}}, \; m, \; f, \; et \right) = [B_{S_1} + B_{S_2} \; , \; et_2]$$

6.6 References and Objects Translation

In the following, we present the translation rules for *cast*, *constructor*, *field-read* and *field-write* sentences.

6.6.1 Cast

Let $B_e, b_{checkcast}, b_{pop}$ be

$$\begin{array}{rl} B_e = & \nabla_E (\; \{ \; \tau(e) \; \preceq \; T \wedge \; P_n[e/x] \; \}, \;\; e, \; \{ \tau(e) \; \preceq \; T \wedge \; shift(P_n[e/x]) \; \wedge \; s(0) = e \; \} \; , \; m[start] \;) \\ b_{checkcast} = & \{ \; \tau(e) \; \preceq \; T \; \wedge \; (shift(P_n[e/x]) \; \wedge \; s(0) = e) \; \} \quad l_b : \textsf{checkcast } T \\ b_{pop} = & \{ \; \tau(e) \; \preceq \; T \; \wedge \; (shift(P_n[e/x]) \; \wedge \; s(0) = e) \; \} \quad l_c : \textsf{pop } x \end{array}$$

$$\nabla_S \left(\frac{}{\left\{ \begin{array}{l} (\; \tau(e) \; \preceq \; T \wedge \; P_n[e/x] \;) \vee \\ (\; \tau(e) \; \not\preceq \; T \wedge \\ Q_e \left[\begin{array}{l} \$ < CastExc > / \$, \\ new(\$, CastExc)/excV \end{array} \right] \;) \end{array} \right\} \quad x = (T) \; e \quad \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & P_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & false) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e) \end{array} \right\}}, \; m, \; f, \; et \right) =$$

$$[\; B_e + b_{checkcast} + b_{pop} \; , \; et \;]$$

6.6.2 New Object

Let b_{newobj}, b_{pop} be

$$\begin{array}{rll} b_{newobj} = & \{P_n[new(\$,T)/x, \$ < T > /\$]\} & l_a : \mathsf{newobj}\ T \\ b_{pop} = & \{(shift(P_n)[s(0)/x])\} & l_b : \mathsf{pop}\ x \end{array}$$

$$\nabla_S \left(\frac{}{\left\{\ P_n[new(\$,T)/x, \$ < T > /\$]\ \right\}\ \ x = new\ T()\ \left\{ \begin{array}{ll} (\mathcal{X} = normal & \Rightarrow P_n) \wedge \\ (\mathcal{X} = break & \Rightarrow false) \wedge \\ (\mathcal{X} = exc & \Rightarrow false) \end{array} \right\}}, m, f, et \right) =$$

$$[\ b_{newobj} + b_{pop}\ ,\ et\ ]$$

6.6.3 Read Field

Let S be the following precondition:

$$S \equiv \left\{ \begin{array}{l} (y \neq null\ \wedge\ P_n[\$(instvar(y, S@a))/x]) \vee \\ (y = null\ \wedge\ Q \left[\begin{array}{l} \$ < NullPExc > /\$, \\ new(\$, NullPExc)/excV \end{array} \right]) \end{array} \right\}$$

Let $b_{push}, b_{getfield}, b_{pop}$ be

$$\begin{array}{rll} b_{push} = & \{y \neq null\ \wedge\ P_n[\$(iv(y, S@a))/x]\} & l_a : \mathsf{pushv}\ y \\ b_{getfield} = & \{s(0) = y\ \wedge Shift(P_n[\$(iv(y, S@a))/x])\} & l_b : \mathsf{getfield}\ S@a \\ b_{pop} = & \{s(0) = \$(iv(y, S@a))\ \wedge Shift(P_n[\$(iv(y, S@a))/x])\} & l_c : \mathsf{pop}\ x \end{array}$$

$$\nabla_S \left(\frac{}{\left\{\ S\ \right\}\ \ x = y.S@a\ \left\{ \begin{array}{ll} (\mathcal{X} = normal & \Rightarrow P_n) \wedge \\ (\mathcal{X} = break & \Rightarrow false) \wedge \\ (\mathcal{X} = exc & \Rightarrow Q) \end{array} \right\}}, m, f, et \right) =$$

$$[\ b_{push} + b_{getfield} + b_{pop}\ ,\ et\ ]$$

6.6.4 Write Field

Let S be the following precondition:

$$S \equiv \left\{ \begin{array}{l} (y \neq null \ \wedge \ P_n[\$ < instvar(y, S@a) := e > /\$]) \ \vee \\ (y = null \ \wedge \ Q \left[\begin{array}{l} \$ < NullPExc > /\$, \\ new(\$, NullPExc)/excV \end{array} \right]) \end{array} \right\}$$

Let $b_{pushv}, b_e, b_{putfield}$ be

$$\begin{array}{rll} b_{pushv} = & \{y \neq null \ \wedge \ P_n[\$ < iv(y, S@a) := e > /\$]\} & l_a : \textsf{pushv } y \\ B_e = & \nabla_E(\{s(0) = y \ \wedge \ shift(P_n[\$ < iv(y, S@a) := e > /\$])\}, \ e, & \\ & \{s(1) = y \ \wedge \ s(0) = e \ \wedge \ shift^2(P_n[\$ < iv(y, S@a) := e > /\$])\} \ , \ l_b) & \\ b_{putfield} = & \{s(1) = y \ \wedge \ s(0) = e \ \wedge \ shift^2(P_n[\$ < iv(y, S@a) := e > /\$])\} & l_c : \textsf{putfield } S@a \end{array}$$

$$\nabla_S \left(\frac{}{\left\{ \ S \ \right\} \quad y.S@a := e \ \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & P_n) \ \wedge \\ (\mathcal{X} = break & \Rightarrow & false) \ \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q) \end{array} \right\}}, \ m, \ f, \ et \right) =$$

$$[\ b_{pushv} + B_e + b_{putfield} \ , \ et \ ]$$

6.6.5 Invocation

Let S be the following precondition:

$$S \equiv \left\{ \begin{array}{l} (y \neq null \ \wedge \ P_n[y/this, e/p]) \ \vee \\ (y = null \ \wedge \ Q_r \left[\begin{array}{l} \$ < NullPExc > /\$, \\ new(\$, NullPExc)/excV \end{array} \right] \ ) \end{array} \right\}$$

Let $b_{pushv}, b_{invokevirtual}, B_e, b_{pop}$ be

$$\begin{array}{rll} b_{pushv} = & \{y \neq null \ \wedge \ P_n[y/this, e/p]\} & l_a : \textsf{pushv } y \\ B_e = & \nabla_E(\{shift(P_n[y/this, e/p]) \ \wedge \ s(0) = y\}, \ e, & \\ & \{shift^2(P_n[y/this, e/p]) \ \wedge \ s(1) = y \ \wedge \ s(0) = e\} \ , \ l_b) & \\ b_{invokevirtual} = & \{shift^2(P_n[y/this, e/p]) \ \wedge \ s(1) = y \ \wedge \ s(0) = e\} & l_c : \textsf{invokevirtual } T : m \\ b_{pop} = & \{Q_n[s(0)/result]\} & l_d : \textsf{pop } x \end{array}$$

$$\nabla_S \left(\frac{\dfrac{Tree_1}{\left\{ \ P_n \ \right\} \quad T : m(p) \ \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & Q_n) \ \wedge \\ (\mathcal{X} = break & \Rightarrow & false) \ \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_r) \end{array} \right\}}}{\left\{ \ S \ \right\} \quad x = y.T : m(e) \ \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & Q_n[x/result]) \ \wedge \\ (\mathcal{X} = break & \Rightarrow & false) \ \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_r) \end{array} \right\}}, \ m, \ f, \ et \right) =$$

$$[\ b_{pushv} + b_{invokevirtual} + B_e + b_{pop} \ , \ et \ ]$$

6.7 Methods

In this section we present the translation for methods which includes proof translation of body methods, and class and subtype rules.

6.7.1 Implementation rule

Let S be the following postcondition:

$$S \equiv \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & Q_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & false) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e) \end{array} \right\}$$

Let B_S, b_{return} be

$$\begin{array}{rl} [B_S, et_2] = & \nabla_S \left(\dfrac{Tree_1}{\left\{ \; P_n \; \right\} \; body(T@m) \; \left\{ \; S \; \right\}}, \; m \left[\; \texttt{next} \; \rightarrow l_b \; \right], \; f, \; et \right) \\ b_{return} = & \{ \; Q_n \; \} \;\; l_b : \texttt{return} \end{array}$$

$$\nabla_S \left(\dfrac{\dfrac{Tree_1}{\left\{ \; P_n \; \right\} \; body(T@m) \; \left\{ \; S \; \right\}}}{\left\{ \; P_n \; \right\} \; T@m \; \left\{ \; S \; \right\}}, \; m, \; f, \; et \right) =$$

$$[\; B_S + b_{return} \; , \; et_2 \;]$$

$$\text{where} \quad \forall i : i < \#(body(T@m)) : body(T@m)_i \neq \texttt{return}$$

6.7.2 Class rule

Let S be the following postcondition:

$$S \equiv \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & Q_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & false) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e) \end{array} \right\}$$

Let B_S, b_{return} be

$$\begin{array}{rl} [B_S, et_2] = & \nabla_S \left(\dfrac{Tree_1}{\left\{ \; \tau(this) = T \; \wedge \; P \; \right\} \; impl(T, m) \; \left\{ \; S \; \right\}}, \; m \left[\; \texttt{next} \; \rightarrow l_b \; \right], \; f, \; et \right) \\ b_{return} = & \{ \; Q_n \; \} \;\; l_b : \texttt{return} \end{array}$$

$$\nabla_S\left(\frac{\dfrac{Tree_1}{\{\ \tau(this) = T\ \wedge\ P\ \}\ \ impl(T,m)\ \ \{\ S\ \}}\qquad\dfrac{Tree_2}{\{\ \tau(this) \prec T\ \wedge\ P\ \}\ \ T:m\ \ \{\ S\ \}}}{\{\ P\ \}\ \ T:m\ \ \{\ S\ \}},\ m,\ f,\ et\right) =$$

$$[\ B_S + b_{return}\ ,\ et_2\ ]$$

6.8 While and Break statements Translation

The use of *break* statements in loops allow us to transfer the control flow to the end of the loop. In this section we discuss about the use of *break* instructions in loops.

In the source logic, we use the variable $\mathcal{X}$ to express the current status of the program. We use it in a loop to express that a break instruction was executed and then we do not change the postcondition because a break was executed. For example:

```
{ true }
while (i<20) {
    { i < 20 }
    if (b) {
        { b ∧ i < 20 }
        i = 40;
        { b ∧ i=40 }
        break;
        { 𝒳 =break ⇒ b ∧ i=40 }
    }
    { (𝒳 = normal ⇒ i < 20) ∧ (𝒳 =break ⇒ b ∧ i=40 ) } (*)
    i = i+1
    { (𝒳 = normal ⇒ i ≤ 20) ∧ (𝒳 =break ⇒ b ∧ i=40 ) } (**)
}
{ (𝒳 = normal ⇒ i = 20) ∧ (𝒳 =break ⇒ b ∧ i=40 ) }
```

When we prove (*) `i = i+1` (**) we keep the postcondition $i = 40$ because we would not execute `i = i+1` if we have executed a break.

It does not happen on bytecode level. We translate the break by a goto and we do not need an extra variable to express that a break has been executed (because we use the goto and we jump to the end of the loop). So, we always can consider normal execution on the bytecode side and we can remove the variable $\mathcal{X}$.

6.8.1 Break statement

When a *break* statement is found, first we translate the proof tree of finally clauses by using the list of $Finally\ f$ (if there is no finally cause, it is empty). Then, we transfer the control flow to the end of the loop by using the mapping function m. f is a list of tuple $[ProofTree\ ,\ ExceptionTable]$, $f = f_1 + ... + f_k$ where $f_i = [t_i, et_i]$ and

$$t_i = \frac{Tree_i}{\{\ A^i\ \}\ \ s_i\ \ \{\ B^i\ \}}$$

Let

$$\begin{aligned}\forall\ f_i =&\ \ [t_i, et_i]\ \in\ f\ (1..k)\ be:\\ [B_i, et'_i] =&\ \ \nabla_S\left(t_i,\ m_i\begin{bmatrix} start & \to l_{a_i}\\ next & \to l_{a_{i+1}}\end{bmatrix},\ f_{i+1} + ... + f_k,\ divide(et'_{i-1}, et_i[0]^1, l_{a_i}, l_{a_i+1})\right)\\ b_{end} =&\ \ \{B^k_b\}\quad l_{k+1} : \text{goto } m[\text{break}]\end{aligned}$$

[1] $et_i[0]$ returns the first exception line of the exception table et_i.

$$\nabla_S \left(\frac{}{\left\{\ P_n\ \right\}\ \ break;\ \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & false) \wedge \\ (\mathcal{X} = break & \Rightarrow & P_n) \wedge \\ (\mathcal{X} = exc & \Rightarrow & false) \end{array} \right\}},\ m,\ f,\ et'_0 \right) =$$

$$[\ B_1 + B_2 + ...B_k + b_{end}\ ,\ \ et'_k\ ]$$

where *divide* is a function: $ExceptionTable \times ExceptionLine \times Label \times Label \rightarrow ExceptionTable$ that divides the exception table taken as first parameter using the exception line table and the starting and finishing labels. It assume that the exception line is included in the exception table. The definition is the following:

$$\begin{array}{lll}
\multicolumn{3}{l}{divide : ExceptionTable \times ExceptionLine\ \times Label \times Label \rightarrow ExceptionTable} \\
divide: & (\ [\ ],\ y,\ l_s,\ l_e) & = [\ y\ ] \\
divide: & (\ x : xs,\ y,\ l_s,\ l_e) & = [\ x_{from},\ l_s,\ x_{targ},\ x_{type}\ ]\ + [\ l_e,\ x_{to},\ x_{targ},\ x_{type}\ ]+ \\
& & \quad divide(xs, y, l_s, l_e)\ \ \textbf{if}\ x \subseteq y\ \wedge\ x \neq y \\
& & \mid x : xs\ \ \textbf{if}\ x = y \\
& & \mid x : divide(xs, y, l_s, l_e)\ \ \textbf{otherwise}
\end{array}$$

where

$$\begin{array}{lll}
\multicolumn{3}{l}{\subseteq : ExceptionLine \times ExceptionLine \rightarrow Boolean} \\
\subseteq: & ([x_{from}, x_{to}, x_{targ}, x_{type}], [y_{from}, y_{to}, y_{targ}, y_{type}]) & = true\ \textbf{if}\ (y_{from} \leq x_{from}) \\
& & \quad \wedge\ (y_{to} \geq x_{to}) \\
& & \mid\ false\ \textbf{otherwise}
\end{array}$$

and

$$\begin{array}{lll}
\multicolumn{3}{l}{= : ExceptionLine \times ExceptionLine \rightarrow Boolean} \\
=: & ([x_{from}, x_{to}, x_{targ}, x_{type}], [y_{from}, y_{to}, y_{targ}, y_{type}]) & = true\ \textbf{if}\ (y_{from} = x_{from})\ \wedge\ (y_{to} = x_{to}) \\
& & \quad \wedge\ (y_{targ} = x_{targ}) \wedge\ (y_{type} = x_{type}) \\
& & \mid\ false\ \textbf{otherwise}
\end{array}$$

Lemma 1

$$\begin{array}{l}
[(I_{l_a}...I_{l_b}),\ et'] = \nabla_S\left(\{P_n\}\ s\ \{Q\},\ m, f, et\ \right) \quad \wedge \quad l_{start} \leq l_a < l_b \leq l_{end} \\
\Rightarrow \\
\forall\ l_s, l_e : l_b < l_s < l_e \leq l_{end} : \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : et[l_{start}, l_{end}, T] = et'[l_s, l_e, T])
\end{array}$$

Lemma 2

Let $et = divide(et_1, r, l_s, l_e)$. *Let* $r = [l_a, l_b, l_t, T]\ \forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any))$.
$(r \subseteq et_1) \Rightarrow et[l_s, l_e, T] = r[T]$

The proofs of lemma 1 and 2 are presented in appendix A.

6.8.2 Translation for while considering break statement and exceptions

Let:

$$
\begin{aligned}
b_{goto} = &\quad \{I_n\} \qquad\qquad l_a : \mathsf{goto}\ l_c \\
[B_{S_1}, et_1] = &\quad \nabla_S \left(\frac{Tree_1}{\left\{ \ e \ \wedge \ I_n \ \right\} \ s_1 \ \left\{ \begin{array}{ll} (\mathcal{X} = normal & \Rightarrow \ I_n) \ \wedge \\ (\mathcal{X} = break & \Rightarrow \ Q_b) \ \wedge \\ (\mathcal{X} = exc & \Rightarrow \ R_e) \end{array} \right\}}, \ m \left[\begin{array}{ll} \mathsf{start} & \rightarrow l_b \\ \mathsf{next} & \rightarrow l_c \\ \mathsf{break} & \rightarrow m[next] \end{array} \right], \emptyset, et \right) \\
B_e = &\quad \nabla_E \ (\ I, \ e, \ (shift(I) \ \wedge \ s(0) = e) \ , l_c \ ) \\
b_{brtrue} = &\quad \{shift(I_n) \ \wedge \ s(0) = e\} \quad l_d : \ \mathsf{brtrue}\ l_b
\end{aligned}
$$

$$
\nabla_S \left(\frac{\dfrac{Tree_1}{\left\{ \ e \ \wedge \ I_n \ \right\} \ s_1 \ \left\{ \begin{array}{ll} (\mathcal{X} = normal & \Rightarrow \ I_n) \ \wedge \\ (\mathcal{X} = break & \Rightarrow \ Q_b) \ \wedge \\ (\mathcal{X} = exc & \Rightarrow \ R_e) \end{array} \right\}}}{\left\{ \ I_n \ \right\} \ while \ (e) \ s_1 \ \left\{ \begin{array}{ll} (\mathcal{X} = normal & \Rightarrow \ (I_n \wedge \neg e) \vee \ Q_b) \ \wedge \\ (\mathcal{X} = break & \Rightarrow \ false) \ \wedge \\ (\mathcal{X} = exc & \Rightarrow \ R_e) \end{array} \right\}}, \ \ m, \ f, \ et \right) =
$$

$$
[\ b_{goto} + B_{S_1} + B_e + b_{brtrue} \ , \ et_1 \ ]
$$

6.9 Exception Handling Translation

Exceptions are exceptional states of a program giving rise to some non-normal mode of execution. In the logic of the source language, exceptions are modeled by using a program variable $\mathcal{X}$ that express the current status of the program.

In the bytecode logic, we define an exception table and variables to express whether an exception was thrown or not. Let *excV* be the value of the exception *exc* and $\tau(excV)$ the type of exception *exc*. When we translate either try catch statements or finally statements, we need to build the exception table. The exception table map an exception type to source and destination labels. We build the exception table at the end of each method.

6.9.1 Try catch translation

Let T_{S_1} be the following proof tree:

$$
T_{S_1} \equiv \frac{Tree_1}{\left\{ \ P_n \ \right\} \ s_1 \ \left\{ \begin{array}{ll} (\mathcal{X} = normal & \Rightarrow \ Q_n) \ \wedge \\ (\mathcal{X} = break & \Rightarrow \ Q_b) \ \wedge \\ (\mathcal{X} = exc & \Rightarrow \ \left(\begin{array}{l} (Q_e \ \wedge \ \tau(excV) \npreceq T) \vee \\ (Q'_e \ \wedge \ \tau(excV) \preceq T) \end{array} \right)) \end{array} \right\}}
$$

and T_{S_2} be

$$T_{S_2} \equiv \frac{Tree_2}{\left\{\ Q'_e[e/excV]\ \right\}\ \ s_2\ \left\{\begin{array}{ll}(\mathcal{X} = normal & \Rightarrow\ Q_n)\ \wedge\\ (\mathcal{X} = break & \Rightarrow\ Q_b)\ \wedge\\ (\mathcal{X} = exc & \Rightarrow\ R_e)\end{array}\right\}}$$

Let:

$$\begin{aligned}
[B_{S_1}, et_1] =\ & \nabla_S\ \Big(T_{S_1},\ m\Big[\ \mathsf{next}\ \ \rightarrow l_b\ \Big],\ f,\ et + [m[start], l_b, l_c, T]\Big)\\
b_{goto} =\ & \{Q_n\} \qquad\qquad\qquad l_b : \mathsf{goto}\ m[next]
\end{aligned}$$

$$b_{pop} = \left\{\begin{array}{l} shift(Q'_e)\ \wedge\\ excV \neq null\\ \wedge\ \tau(excV) \preceq T\\ \wedge\ s(0) = excV\end{array}\right\}\quad l_c : \mathsf{pop}\ e$$

$$[B_{S_2}, et_2] =\ \nabla_S\ \Big(T_{S_2},\ m\Big[\ \mathsf{start}\ \ \rightarrow l_d\ \Big],\ f,\ et_1\Big)$$

$$\nabla_S\left(\frac{T_{S_1}\qquad\qquad T_{S_2}}{\left\{\ P_n\ \right\}\ \ try\ s_1\ catch\ (T\ e)\ s_2\ \left\{\begin{array}{ll}(\mathcal{X} = normal & \Rightarrow\ Q_n)\ \wedge\\ (\mathcal{X} = break & \Rightarrow\ Q_b)\ \wedge\\ (\mathcal{X} = exc & \Rightarrow\ \left(\begin{array}{l} R_e\ \vee\\ (Q_e\ \wedge\ \tau(excV) \npreceq T)\end{array}\right))\end{array}\right\}},\ m,\ f,\ et\right) =$$

$$[\ B_{S_1} + b_{goto} + b_{pop} + B_{S_2}\ ,\ et_2\ ]$$

6.9.2 Throw Translation

Let b_1 and b_2 be

$$\begin{aligned}
B_e &= \nabla_E\Big(\ P_n[e/excV]\ ,\ e,\ \ (shift(P_n[e/excV])\ \wedge\ s(0) = e)\ ,\ m[start]\ \Big)\\
b_{athrow} &= \{\ shift(P_n[e/excV])\ \wedge\ s(0) = e\ \}\quad l_b : \mathsf{athrow}
\end{aligned}$$

$$\nabla_S\left(\frac{}{\left\{\ P_n[e/excV]\ \right\}\ \ throw\ e\ \left\{\begin{array}{ll}(\mathcal{X} = normal & \Rightarrow\ false)\ \wedge\\ (\mathcal{X} = break & \Rightarrow\ false)\ \wedge\\ (\mathcal{X} = exc & \Rightarrow\ P_n)\end{array}\right\}}\ ,\ m,\ f,\ et\right) = [\ B_e + b_{athrow}\ ,\ et\ ]$$

We assume that athrow instruction takes the expression e on the top of the stack, then assigns it to the variable $excV$ and finally jumps to the label where the exception is caught (it is defined by the exception table).

6.9.3 Translation of Finally statements for Java

We can translate the finally statements in two different ways. The first way is to use the call of subroutines and the instruction jsr. The second one is to use code duplication. We use code duplication to simplify the translation. Also, we generate the same code than the **javac** compiler.

Let S, T, U, V be

$$S \equiv \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & Q_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & Q_b) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e) \end{array} \right\}$$

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

$$U \equiv \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & \left(\begin{array}{l} (Q'_n \ \wedge \ \mathcal{X}Tmp = normal) \vee \\ (Q'_b \ \wedge \ \mathcal{X}Tmp = break) \vee \\ (Q'_e \ \wedge \ \mathcal{X}Tmp = exc) \end{array} \right)) \wedge \\ (\mathcal{X} = break & \Rightarrow & R_b) \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q'_e) \end{array} \right\}$$

$$V \equiv \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & Q'_n \ ) \ \wedge \\ \quad (\mathcal{X} = break & \Rightarrow & (Q'_b \ \vee \ R_b) \ ) \wedge \\ \quad (\ \mathcal{X} = exc & \Rightarrow & Q'_e \ ) \end{array} \right\}$$

Let m_1 be an mapping function, and et', et'' be exception tables

$$\begin{array}{l} m_1 \equiv m \left[\ \mathsf{next} \quad \rightarrow l_b \ \right] \\ et' = et + [l_a, l_b, l_d, any] \\ et'' = getExceptions(l_a, l_b, et')^2 \end{array}$$

[2] getExceptions returns the exceptions of the exception table et' between l_a and l_b. It returns at most one exception line for every type.

$$[B_{S_1}, et_1] = \nabla_S \left(\frac{Tree_1}{\left\{ P_n \right\} \ s_1 \ \left\{ S \right\}}, m_1 , \left[\frac{Tree_2}{\left\{ T \right\} \ s_2 \ \left\{ U \right\}} , et'' \right] + f, \ et' \right)$$

$$[B_{S_2}, et_2] = \nabla_S \left(\frac{Tree_2}{\left\{ T \right\} \ s_2 \ \left\{ U \right\}}, m \begin{bmatrix} \textsf{start} & \rightarrow l_b \\ \textsf{next} & \rightarrow l_c \end{bmatrix}, \ f, \ et_1 \right)$$

$$b_{goto} = \{Q'_n\} \qquad l_c : \textsf{goto } m[next]$$

$$b_{pop} = \left\{ \begin{array}{l} shift(Q_e) \wedge \\ excV \neq null \\ \wedge \ s(0) = excV \end{array} \right\} \qquad l_d : \textsf{pop } eTmp$$

$$[B'_{S_2}, et_3] = \nabla_S \left(\frac{Tree_2}{\left\{ T \right\} \ s_2 \ \left\{ U \right\}}, m \begin{bmatrix} \textsf{start} & \rightarrow l_e \\ \textsf{next} & \rightarrow l_f \end{bmatrix}, \ f, \ et_2 \right)$$

$$b_{pushv} = \left\{ \ Q'_n \ \vee \ Q'_b \ \vee \ Q'_e \ \right\} \qquad l_f : \textsf{pushv } eTmp$$

$$b_{athrow} = \left\{ \begin{array}{l} (Q'_n \ \vee \ Q'_b \ \vee \ Q'_e) \\ \wedge \ s(0) = eTmp \end{array} \right\} \qquad l_g : \textsf{athrow}$$

$$\nabla_S \left(\frac{\dfrac{Tree_1}{\left\{ P_n \right\} \ s_1 \ \left\{ S \right\}} \qquad \dfrac{Tree_2}{\left\{ T \right\} \ s_2 \ \left\{ U \right\}}}{\left\{ P_n \right\} \ try \ s_1 \ finally \ s_2 \ \left\{ V \right\}}, \ m, \ f, \ et \right) =$$

$$[\ B_{S_1} + B_{S_2} + b_{goto} + b_{pop} + B'_{S_2} + b_{pushv} + b_{athrow} \ , \ et_3 \ ]$$

7 Application

In this section we present two examples that translate a source object-oriented program and a proof to bytecode with its bytecode proof. The first example (section 7.1) is the translation of example of table 1 presented in section 3.7.1 on page 15 . The second example (section 7.2) is the translation of the example of table 2 presented in section 3.7.2 on page 16.

7.1 Translation of while and break statements

In this section we present the translation from the proof of figure 1 to bytecode. To simplify the proof, we suppose that the array a of length n is equal to n variables a_0, a_1, ... a_{n-1}. Then we do not use the special instruction to load arrays and we use instructions such as pushv a_0 or pushv a_{ind} where ind is the index of the array. The code and proof for pushv a_{ind} is the following:

{ $0 \leq$ ind $<$ sizeof(a) }	l_{00} : pushv a \\ a is the array
{ s(0)=a $\wedge$ $0 \leq$ ind $<$ sizeof(a)}	l_{01} : pushv ind \\ ind is a index of an array
{ s(1)=a $\wedge$ $s(0) = ind$ $\wedge$ $0 \leq$ ind $<$ sizeof(a) }	l_{02} : iaload \\ load the value of the array a in the index ind

Furthermore, the instruction pushv $a.length$ is not present on the bytecode language. We also use it to simplify the example. The real code and proof for this instruction is the following:

$$\begin{array}{ll} \{\ a \neq null\ \} & l_{00} : \texttt{pushv a} \quad \text{\\\\ a is the array} \\ \{\ \text{s(0)=a}\ \} & l_{01} : \texttt{arraylength} \quad \text{\\\\ get the length of an array} \end{array}$$

7.1.1 Bytecode Proof

To check that a proof is a valid proof in the bytecode logic, we need to verify that each instruction implies the local weakest precondition. For example, we have to prove:

$$\frac{\{ind < a.length \ \wedge \ I \ \wedge \ a_{ind} \neq sum\} \Rightarrow wp_p^1(l_{b06} : \ \texttt{pushc}\ False)}{\left\{ \begin{array}{l} ind < a.length \ \wedge \ I \ \wedge \\ a_{ind} \neq sum \end{array} \right\} \quad l_{b06} : \texttt{pushc}\ False \quad \left\{ \begin{array}{l} ind < a.length \ \wedge \ I \ \wedge \\ s(0) = False \ \wedge \\ a_{ind} \neq sum \end{array} \right\}}$$

By wp_p^1 definition (figure 5) we get

$$\begin{array}{l} \{ind < a.length \ \wedge \ I \ \wedge \ a_{ind} \neq sum\} \\ \quad \Rightarrow \\ unshift(ind < a.length \ \wedge \ I \ \wedge \ s(0) = False \ \wedge \ a_{ind} \neq sum[False/s(0)]) \\ \quad \Rightarrow \ [| \ by\ replacement \ |] \\ unshift(ind < a.length \ \wedge \ I \ \wedge \ False = False \ \wedge \ a_{ind} \neq sum) \\ \quad \Rightarrow \ [| \ by\ def\ unshift \ |] \\ ind < a.length \ \wedge \ I \ \wedge \ False = False \ \wedge \ a_{ind} \neq sum) \\ \square \end{array}$$

7.2 Application of try catch and throw statements

Figure 7 presents the generated bytecode for the source program of figure 3. We present the bytecode proof in figure 8.

$\{a \neq null \ \wedge \ a.length > 0\}$	l_0 : pushc 1
$\{a \neq null \ \wedge \ a.length > 0 \ \wedge \ s(0) = 1\}$	l_1 : pop ind
$\{a \neq null \ \wedge \ a.length > 0 \ \wedge \ ind = 1\}$	l_2 : pushv a_0
$\{a \neq null \ \wedge \ a.length > 0 \ \wedge \ ind = 1 \wedge \ s(0) = a_0\}$	l_3 : pop sum
$\{a \neq null \ \wedge \ a.length > 0 \ \wedge \ ind = 1 \wedge \ sum = a_0\}$	l_4 : pushc $True$
$\{a \neq null \ \wedge \ a.length > 0 \ \wedge \ ind = 1 \wedge \ sum = a_0 \ \wedge \ s(0) = True\}$	l_5 : pop $result$
$\Vert$ **begin body while (ind $<$ a.lenght)**	
$\{I\ \}$	l_a : goto l_{c1}
$\Vert$ **if a[ind] ! = sum then**	
$\{(ind < a.length \ \wedge \ I\ \}$	l_{b01} : pushv a_{ind}
$\{shift(ind < a.length \ \wedge \ I) \wedge s(0) = a_{ind}\}$	l_{b02} : pushv sum
$\{shift^2(ind < a.length \ \wedge \ I) \wedge s(1) = a_{ind} \wedge \ s(0) = sum\}$	l_{b03} : $binop_{\neq}$
$\{shift(ind < a.length \ \wedge \ I) \ \wedge \ s(0) = (a_{ind} \neq sum)\}$	l_{b04} : brtrue l_{b6}
$\{ind < a.length \ \wedge \ I \ \wedge \ a_{ind} = sum\}$	l_{b05} : goto l_{b11}
$\{ind < a.length \ \wedge \ I \ \wedge \ a_{ind} \neq sum\}$	l_{b06} : pushc $False$
$\{ind < a.length \ \wedge \ I \ \wedge \ s(0) = False \ \wedge \ a_{ind} \neq sum\}$	l_{b07} : pop $result$
$\{ind < a.length \ \wedge \ I \ \wedge \ result = False \ \wedge \ a_{ind} \neq sum\}$	l_{b08} : nop
$\{result = False \wedge \ (\exists\ j :\ 0 \leq j < a.length :\ (a[j] \neq \sum_{k=0}^{j-1} \ a[k])\ )\}$	l_{b10} : goto l_g
$\Vert$ **end a[ind] ! = sum**	
$\Vert$ **sum = sum + a[ind]; ind = ind + 1**	
$\{(ind < a.length \ \wedge \ I \ \wedge \ a_{ind} = \sum_{k=0}^{ind-1} a_k) \equiv P_2\}$	l_{b11} : pushv a_{ind}
$\{shift(P_2) \ \wedge \ s(0) = a_{ind}\}$	l_{b12} : pushv sum
$\{shift^2(P_2) \ \wedge \ s(1) = a_{ind} \ \wedge \ s(0) = sum\}$	l_{b13} : $binop_{+}$
$\{shift(P_2) \ \wedge \ s(0) = a_{ind} + sum\}$	l_{b14} : pop sum
$\{\ Q\ \}$	l_{b15} : pushc 1
$\{\ shift(Q) \ \wedge \ s(0) = 1\}$	l_{b16} : pushv ind
$\{\ shift^2(Q) \ \wedge \ s(1) = 1 \ \wedge s(0) = ind\}$	l_{b17} : $binop_{+}$
$\{\ shift(Q) \ \wedge \ s(0) = 1 + ind\}$	l_{b18} : pop ind
$\Vert$ **end a[ind] = sum**	
$\{\ I\ \}$	l_{c1} : pushv ind
$\{shift(I) \ \wedge \ s(0) = ind\}$	l_{c2} : pushv $a.length$
$\{shift^2(I) \ \wedge \ s(1) = ind \ \wedge \ s(0) = a.length\}$	l_{c3} : $binop_{<}$
$\{shift(I) \ \wedge \ s(0) = (ind < a.length)\}$	l_d : brtrue l_{b01}
$\Vert$ **end body while**	
$\{R\ \}$	l_g : pushv $result$
$\{R \ \wedge \ s(0) = result\ \}$	l_h : areturn

$$where \quad I \equiv result \ \wedge \ (\forall\ j :\ 0 \leq j < ind :\ (\ a[j] = \sum_{k=0}^{j-1} a[k]\ )) \\ \wedge\ sum = \sum_{k=0}^{ind-1} a[k] \ \wedge \ 0 \leq ind \leq a.length)\ \}$$

$$and \qquad Q \equiv \left(\begin{array}{l} ind < a.length \ \wedge \ a[ind] = \sum_{k=0}^{ind-1} a[k] \ \wedge sum = (\sum_{k=0}^{ind-1} a[k]) + a[ind] \ \wedge \\ result \ \wedge \ (\forall\ j :\ 0 \leq j < ind :\ (\ a[j] = \sum_{k=0}^{j-1} a[k]\ )) \wedge \ 0 \leq ind \leq a.length \end{array} \right)$$

$$and \qquad R \equiv \left(\begin{array}{l} (result \ \wedge \ \forall\ j :\ 0 \leq j < a.length :\ (\ a[j] = \sum_{k=0}^{j-1} a[k]\ )) \ \vee \\ (result = False \wedge \ \exists\ j :\ 0 \leq j < a.length : (a[j] \neq \sum_{k=0}^{j-1} \ a[k])) \end{array} \right)$$

Figure 6: Bytecode proof of the source proof of table 1.

```
public int exampleTry(boolean b);
  Code:
   0:   pushv b
   1:   invokevirtual myMet
   4:   pop auxExc
   5:   pushc 0
   6:   pop result
   7:   goto 13
   10:  pop e
   11:  pushv 1
   12:  pop result
   13:  pushv result
   14:  areturn
  Exception table:
   from   to   target  type
     0     7     10    Class java/lang/Exception

public java.lang.Exception myMet(boolean);
   throws java/io/IOException, java/lang/Exception
  Code:
   0:   pushv b
   1:   brfalse   12
   4:   newobj java/lang/Exception
   11:  athrow
   12:  newobj java/lang/Exception
   19:  areturn
```

Figure 7: Bytecode of the example figure 3.

7.3 Application of try-finally, while and break

```
foo () {
    int b;
    { true }
    b=1;
    { b=1, false, false }
    while (true) {
    { b=1, false, false }
        try {
            { b=1, false, false }
            b=b+1;
            { b=2, false, false }
            throw new Exception();
            { false, false, b=2 }
        }
        finally {
            { b=2 and Xtmp=exc }
            b++;
            { b=3 and Xtmp=exc, false, false }
            break;
            { false, b=3 and Xtmp=exc, false }
        }
        { false, b=3, false }
    }
    { b=3,false,false }
    b++;
    { b=4, false, false }
}
```

public int exampleTry(boolean b);
Code:

try

$\{ b = B \wedge OS = \$ \}$	0: pushv b
$\{ b = B \wedge OS = \$ \wedge s(0) = B \}$	1: invokevirtual myMet
$\left\{ \begin{array}{l} s(0) = new(\$, Exception) \wedge \\ \$ = \$ < Exception > \wedge \neg B \end{array} \right\}$	4: pop auxExc
$\left\{ auxExc = new(\$, Exception) \wedge \neg B \right\}$	5: pushc 0
$\left\{ auxExc = new(\$, Exception) \wedge \neg B \wedge s(0) = 0 \right\}$	6: pop result
$\left\{ auxExc = new(\$, Exception) \wedge \neg B \wedge result = 0 \right\}$	7: goto 13

catch (Exception e)

$\left\{ \begin{array}{l} B \wedge excV \neq null \wedge \\ \tau(excV) \preceq Exception \wedge s(0) = excV \end{array} \right\}$	10: pop e
$\left\{ B \wedge e = excV \right\}$	11: pushv 1
$\{ B \wedge s(0) = 1 \}$	12: pop result
$\left\{ \begin{array}{l} (auxExc = new(\$, Exception) \wedge \neg B \wedge result = 0) \vee \\ (B \wedge result = 1) \end{array} \right\}$	13: pushv result
$\left\{ \left(\begin{array}{l} (auxExc = new(\$, Exception) \\ \wedge \neg B \wedge result = 0) \vee \\ (B \wedge result = 1) \end{array} \right) \wedge s(0) = result \right\}$	14: areturn

Exception Table

From	to	target	type
0	7	10	Class java/lang/Exception

public java.lang.Exception myMet(boolean);
    throws java/io/IOException, java/lang/Exception
Code:
requires: $\{b = B \wedge OS = \$\}$

$\{ b = B \wedge OS = \$ \}$	0: pushv b
$\{ b = B \wedge OS = \$ \wedge s(0) = B \}$	1: brfalse 12
$\{ b \wedge OS = \$ \}$	4: newobj java/lang/Exception
$\left\{ \begin{array}{l} (s(0) = new(\$, Exception) \\ \wedge \$ = \$ < Exception > \wedge B) \end{array} \right\}$	11: athrow
$\{ \neg B \wedge OS = \$ \}$	12: newobj java/lang/Exception
$\left\{ \begin{array}{l} (s(0) = new(\$, Exception) \\ \wedge \$ = \$ < Exception > \wedge \neg B) \end{array} \right\}$	19: areturn

ensures:

$$\left\{ \begin{array}{l} (result = new(\$, Exception) \wedge \$ = \$ < Exception > \wedge \neg B) \vee \\ (excV \neq null \wedge B)) \end{array} \right\}$$

Figure 8: Bytecode proof of the example table 2.

$\{true\}$	l_{00} : pushc 1
$\{s(0) = 1\}$	l_{01} : pop b
$\{b = 1\}$	l_{02} : goto l_{03}
$\|\|$ **body while**	
$\{b = 1\}$	l_{10} : pushc 1
$\{b = 1 \ \wedge \ s(0) = 1\}$	l_{11} : pushv b
$\{b = 1 \ \wedge \ s(1) = 1 \ \wedge \ s(0) = b\}$	l_{12} : $binop_{+}$
$\{b = 1 \ \wedge \ s(0) = b + 1\}$	l_{13} : pop b
$\{b = 2\}$	l_{14} : pushc 1
$\{b = 2 \ \wedge \ s(0) = 1\}$	l_{15} : pushv b
$\{b = 2 \ \wedge \ s(1) = 1 \ \wedge \ s(0) = b\}$	l_{16} : $binop_{+}$
$\{b = 2 \ \wedge \ s(0) = b + 1\}$	l_{17} : pop b
$\{b = 3\}$	l_{18} : goto l_{05}
$\{false\}$	l_{19} : newobj $Exception$
$\{false\}$	l_{20} : athrow
$\|\|$ **code duplication finally**	
$\{false\}$	l_{21} : pushc 1
$\{false\}$	l_{22} : pushv b
$\{false\}$	l_{23} : $binop_{+}$
$\{false\}$	l_{24} : pop b
$\{false\}$	l_{25} : pushv $eTmp$
$\{false\}$	l_{26} : athrow
$\{b = 1\}$	l_{03} : pushc $true$
$\{b = 1 \ \wedge \ s(0) = true\}$	l_{04} : brtrue l_{10}
$\|\|$ **endwhile**	
$\{b = 3\}$	l_{05} : pushc 1
$\{b = 3 \ \wedge \ s(0) = 1\}$	l_{06} : pushv b
$\{b = 3 \ \wedge \ s(1) = 1 \ \wedge \ s(0) = b\}$	l_{07} : $binop_{+}$
$\{b = 3 \ \wedge \ s(0) = 1 + b\}$	l_{08} : pop b

8 Soundness of the translation

We prove that the translation is correct. Before present the theorem, let us introduce some notation.

- m is a mapping function: $Event \rightarrow Label$. $m[start]$ returns the label stored in the event $start$.
- Every bytecode instruction specification is written as: $\{E_l\}\ l :\ I_l$ where $\{E_l\}$ is the precondition, l a label and I_l a bytecode instruction. For example, $E_{m[exc\_T]}$ returns the precondition of the instruction at the label $m[exc\_T]$.
- Let $et[l_{start}, l_{end}, T]$ be a function: $ExceptionTable \times Label \times Label \times Type \rightarrow Label$ defined as $et[l_a, l_b, T] = getTargetLabel(et, l_{start}, l_{end}, T)$ where $getTargetLabel$ is defined as following:

$$\begin{array}{lll}
\multicolumn{3}{l}{getTargetLabel : ExceptionTable \times Label \times Label \times Type \rightarrow Label} \\
\ getTargetLabel : & (\ [\ ],\ l_s,\ l_e,\ T\ ) & = l_{end\_method} \\
\ getTargetLabel : & (\ ([l'_s, l'_e, l_t, T'] + et)\ , l_s, l_e, T) & = l_t \quad if\ (l'_s \leq l_s \ \wedge \ l'_e \geq l_e \ \wedge \ T \preceq T') \\
 & & = getTargetLabel(et, l_s, l_e, T) \quad otherwise
\end{array}$$

- Let $et[T]$ be a function: $ExceptionTable \times Type \rightarrow Label$. This is a partial function defined over the exceptions tables that satisfy $(\forall\ et_i = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset))$. Then to get the exception we only use the type. It is defined as following:

$$\begin{array}{lll}
\multicolumn{3}{l}{getTargetLabel : ExceptionTable \times Type \rightarrow Label} \\
\ getTargetLabel : & (\ [\ ],\ T\ ) & = l_{end\_method} \\
\ getTargetLabel : & (\ ([l'_s, l'_e, l_t, T'] + et)\ , T) & = l_t \quad if\ (T \preceq T') \\
 & & = getTargetLabel(et, T) \quad otherwise
\end{array}$$

- f^* is a list of $[ProofTree, ExceptionTable]$ possible empty $f = f_1 + ... + f_k$ where $f_i = [t_i, et_i]$. This list is constructed only in the finally translation (section 6.9.3, on page 40). So we know that f has an special form. We always construct f with proof trees of the form:

$$f_i = \left[\frac{Tree_i}{\{\ A^i\ \}\ \ s_i\ \ \{\ B^i\ \}}, et_i \right] \ \in\ f\ (1..k) :$$

where A^i and B^i are in the normal form:

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

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

The theorem expresses that

- if we have a source proof for the statement s, and
- we have a proof translation from the source proof that produces the instructions $I_{l_{start}}...I_{l_{end}}$ and their respective preconditions $E_{l_{start}}...E_{l_{end}}$, and the exception table et and
- the exceptional postconditon in the source logic implies the precondition at the target label stored in the exception table for all type T such that $((T \preceq Throwable) \vee (T \equiv any))$ but considering the value stored in the stack of the bytecode, and
- 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), and
- if the break postcondition is not false then
  - for every triple stored in f, the triple holds and the break postcondition of the triple (denoted by B^i_b) implies the break precondition of the next triple (denoted by A^{i+1}_b). And the exceptional postconditon B^i_e 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. Furthermore, the break postcondition (Q_b) implies the first break precondition of the before mentioned triples. And the last postcondition of the triple implies the postcondition stored on the mapping function m at the event *break*, and
  - if f is empty then the break postcondition (in the source logic) implies the precondition at the label stored in the mapping function at the type *break* (in the bytecode logic).

then we have to prove that every bytecode specification holds ($\vdash \{E_l\}\ I_l$).
The theorem is the following:

Theorem 1

$$\vdash \frac{Tree_1}{\left\{\ P\ \right\}\ s_1 \left\{\begin{array}{lll}(\mathcal{X} = normal & \Rightarrow & Q_n)\ \wedge\\ (\mathcal{X} = break & \Rightarrow & Q_b)\ \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e)\end{array}\right\}} \quad \wedge$$

$$[(I_{l_{start}} ... I_{l_{end}}),\ et] = \nabla_S \left(\frac{Tree_1}{\left\{\ P\ \right\}\ s_1 \left\{\begin{array}{lll}(\mathcal{X} = normal & \Rightarrow & Q_n)\ \wedge\\ (\mathcal{X} = break & \Rightarrow & Q_b)\ \wedge \\ (\mathcal{X} = exc & \Rightarrow & Q_e)\end{array}\right\}},\ m\left[\begin{array}{ll} start & \to l_{start}\\ next & \to l_{end+1}\end{array}\right], f^*, et' \right) \quad \wedge$$

$$\begin{array}{l}
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (Q_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{et'[l_{start}, l_{end}, T]})\ \wedge \\
(Q_n \ \Rightarrow\ E_{m[next]})\ \wedge \\
\neg(Q_b\ \Rightarrow\ false) \Rightarrow
\end{array}$$

$$\left(\begin{array}{l}
f \neq \emptyset\ \Rightarrow\ \forall i\ \in\ 1..k : \left(\begin{array}{l}
(\vdash \{A^i\}\ s_i\ \{B^i\} \quad \wedge\ \ (B^i_b \Rightarrow A^{i+1}_b)\ \wedge \\
(\ \forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (B^i_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{et_i[T]}\ )\ \wedge \\
(Q_b \Rightarrow\ A^1_b)\ \wedge\ (B^k_b\ \Rightarrow\ E_{m[break]})\ \wedge \\
(\forall\ et_i = r^1_i + r^2_i + ... + r^m_i : (r^1_i \cap r^2_i \cap ... \cap r^m_i = \emptyset)\ \wedge (et_i \subseteq et'))
\end{array}\right) \\
f = \emptyset\ \Rightarrow\ (Q_b\ \Rightarrow\ E_{m[break]})
\end{array}\right)$$

$$\Rightarrow$$

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

The proof is done by induction on the structure of the derivation tree for $\{P\}\ s\ \{Q\}$. We present the proof in the appendix A.

9 Related Work

Necula and Lee [14] have developed certifying compilers, which produce proofs for basic safety properties such as type safety. Since our approach supports interactive verification of source programs, we can handle more complex properties such as functional correctness.

The open verifier framework for foundational verifiers [6] verifies untrusted code using customized verifiers. The approach is based on foundation proof carrying code. The architecture consists of a trusted checker, a fixpoint module, and an untrusted extension (a new verifier developed by untrusted users). However, the properties that can be proved are still limited.

A certified compiler [8, 20] is a compiler that generates a proof that the translation from the source program to the assembly code preserves the semantics of the source program. Together with a source proof, this gives an indirect correctness proof for the bytecode program. Our approach generates the bytecode proof directly, which leads to smaller certificates.

Barthe *et al.* [4] show that proof obligations are preserved by compilation (for a non-optimizer compiler). They prove the equivalence between the verification condition (VC) generated over the source code and the bytecode. The source language is an imperative language which includes method invocation, loops, conditional statements, `try-catch` and `throw` statements. However, they do not consider `try-finally` statements, which make the translation significantly more comples. Our translation supports `try-finally` and `break` statements.

Pavlova [17] extends the aforementioned work to a subset of Java (which includes `try-catch`, `try-finally`, and `return` statements). She proves equivalence between the VC generated from the source program and the VC generated from the bytecode program. The translation of the above source language has a similar complexity to the translation presented in this paper. However, Pavlova avoided the code duplication for `finally` blocks by disallowing `return` statements inside the `try` blocks of `try-finally` statements. This simplifies not only the verification condition generator, but also the translation and the soundness proof.

Furthermore, Barthe *et al.* [5] translate certificates for optimizing compilers from a simple interactive language to an intermediate RTL language (Register Transfer Language). The translation is done in two steps: first the source program is translated into RTL and then optimizations are performed building the appropriate certificate. Barthe *et al.* use a source language that is simpler than ours. We will investigate optimizing compilersas part of future work.

This work is based on Müller and Bannwart's work [**?**]. They present a proof-transforming compiler from a subset of Java which includes loops, conditional statements and object oriented features. We have extended the source language including exception handling and `break` statements. Moreover, we have also proved soundness.

10 Conclusions

We have defined a proof transformation from a subset of C#, Eiffel and Java to Bytecode. The source language includes the most important object-oriented languages features, including methods invocation, subtyping, and exception handling. The PTC allows us to develop the proof in the source language (which is simpler), and transforms it into a bytecode proof.

A proof checker can be used to verify wether the source program is safe or not. The proof checker is the only trusted component. If the source proof or the translation were incorrect then the checker would reject the code.

The generated bytecode is similar to the bytecode generated by javac compiler. Furthermore, we proved that the translation is sound.

As future work, we plan to extend the source language including features such as multiple inheritance, which requires more complex transformations. This extension will lead to a more general transformation framework.

References

[1] F. Y. Bannwart and P. Müller, "A Logic for Bytecode" in proceedings Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE), Electronic Notes in Theoretical Computer Science, 2005.

[2] F. Y. Bannwart and P. Müller, "A Logic for Bytecode". Technical Report 469, ETH Zürich. 2004. Available from sct.inf.ethz.ch/publications.

[3] F. Y. Bannwart, "A Logic for Bytecode and the translation of proof from sequential Java". ETH Zürich. June 2004.

[4] G. Barthe, t. Rezk and A. Saabas, "Proof obligations preserving compilation" in Formal Aspects in Security and Trust, pages 112-126, 2005.

[5] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk, Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, Robert R. Schneck "Certificate translation for optimizing compilers". In Proceedings of the 13th International Static Analysis Symposium (SAS), LNCS Springer-Verlag, Seoul, Korea, 2006.

[6] B. Chang, A. Chlipala, G. Necula, R. Schneck, "The Open Verifier Framework for Foundational Verifiers" in Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI05), 2005.

[7] C. Colby, P. Lee, G. Necula, F. Blau, M. Plesko, K. Cline, "A certifying compiler for Java", in *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'00), pages 95–105, ACM Press, Vancouver (Canada), June 2000.

[8] G. Goos and W. Zimmermann, "Verification of Compilers", Lecture Notes In Computer Science; Vol. 1710, Springer-Verlag, pages 201 - 230. 1999.

[9] B. Meyer, "Object-Oriented Software Construction". Prentice Hall, first edition. 1988.

[10] B. Meyer, "Object-Oriented Software Construction". Prentice Hall, second edition. 1997.

[11] P. Müller, "Modular Specification and verification of object-oriented programs", Springer-Verlag, 2002.

[12] The MSDN library, Microsoft Corporation. `http://msdn2.microsoft.com/en-us/library/0hbbzekw.aspx`.

[13] G. Necula "Compiling with Proofs" Ph.D. Thesis School of Computer Science, Carnegie Mellon University CMU-CS-98-154. 1998.

[14] G. Necula, P. Lee, "The Design and Implementation of a Certifying Compiler", in *Proceedings of the 1988 ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI'98), pages 333–344, ACM Press, Montreal (Canada), June 1998.

[15] D. von Oheimb, "Analyzing Java in Isabelle/HOL - Formalization, Type Safety and Hoare Logic -", Ph.D. thesis, Technische Universitt München. 2001.

[16] D. von Oheimb, "Hoare Logic for Java in Isabelle/HOL", in special issue of Concurrency and Computation: Practice and Experience Volume 13, Issue 13, pages 1173-1214, November 2001.

[17] M. Pavlova, "Java Bytecode verification and its applications", Ph.D. thesis, University of Nice Sophia-Antipolis. 2007.

[18] A. Poetzsch-Heffter and Marek Gawkowski, "Towards Proof Generating Compilers", in Compiler Optimization Meets Compiler Verification (COCV), 2004.

[19] A. Poetzsch-Heffter, and N. Rauch, "Soundness and Relative Completeness of a Programming Logic for a Sequential Java Subset". Technical Report, Technische Universität Kaiserlautern, Germany. February 2004.

[20] A. Poetzsch-Heffter and Marek Gawkowski, "Towards Proof Generating Compilers", in Compiler Optimization Meets Compiler Verification (COCV), 2004.

[21] A. Poetzsch-Heffter, and P. Müller, "A Programming Logic for Sequential Java", in proceedings European Symposium on Programming Languages and Systems (ESOP), pages 162–176, Lecture Notes in Computer Science Springer-Verlag, 1999.

[22] A. Poetzsch-Heffter, and P. Müller, "Logical Foundations for Typed Object-Oriented Languages", in Programming Concepts and Methods (PROCOMET), pages 404–423, Gries, D. and De Roever, W., 1998.

[23] A. Poetzsch-Heffter, "Specification and verification of object-oriented programs ". Habilitation thesis, Technical University of Munich, January 1997.

[24] H. Wasserman and Manuel Blum, "Software Reliability via Run-Time Result-Checking", in Journal of the ACM volume 44, number 6, pages 826-849. 1997.

A Appendix: Soundness proof

In this section we present the soundness proof of the translation. The proof is done by induction on the structure of the derivation tree for $\{P\}\ s\ \{Q\}$. We present the proof for the most important cases but the remaining cases are similar.

A.1 Notation

To make the proof easier to read, we write

$$\nabla_S(\ \{P_n\}\ s_1; s_2\ \{R\},\ m, f)$$

meaning:

$$\nabla_S\left(\frac{T_{S_1} \qquad T_{S_2}}{\left\{\ P\ \right\}\ \ s_1; s_2\ \left\{\begin{array}{lll}(\mathcal{X} = \mathit{normal} & \Rightarrow & R_n) \wedge \\ (\mathcal{X} = \mathit{break} & \Rightarrow & R_b) \wedge \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & R_e)\end{array}\right\}},\ m,\ f\right) =$$

where T_{S_1} and T_{S_2} are the following proof trees:

$$T_{S_1} \equiv \frac{T_1}{\left\{\ P_n\ \right\}\ \ s_1\ \left\{\begin{array}{lll}(\mathcal{X} = \mathit{normal} & \Rightarrow & Q_n) \wedge \\ (\mathcal{X} = \mathit{break} & \Rightarrow & R_b) \wedge \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & R_e)\end{array}\right\}}$$

$$T_{S_2} \equiv \frac{T_2}{\left\{\ Q_n\ \right\}\ \ s_2\ \left\{\begin{array}{lll}(\mathcal{X} = \mathit{normal} & \Rightarrow & R_n) \wedge \\ (\mathcal{X} = \mathit{break} & \Rightarrow & R_b) \wedge \\ (\mathcal{X} = \mathit{exc} & \Rightarrow & R_e)\end{array}\right\}}$$

and we write

$$\vdash \{P_n\}\ s_1; s_2\ \{R\}$$

meaning

$$\vdash \frac{\left\{\ P\ \right\}\ s_1\ \left\{\begin{array}{ll}(\mathcal{X} = \mathit{normal} & \Rightarrow\ Q)\ \wedge \\ (\mathcal{X} = \mathit{break} & \Rightarrow\ R_b)\ \wedge \\ (\mathcal{X} = \mathit{exc} & \Rightarrow\ R_e)\end{array}\right\} \qquad \left\{\ Q\ \right\}\ s_2\ \left\{\begin{array}{ll}(\mathcal{X} = \mathit{normal} & \Rightarrow\ R_n)\ \wedge \\ (\mathcal{X} = \mathit{break} & \Rightarrow\ R_b)\ \wedge \\ (\mathcal{X} = \mathit{exc} & \Rightarrow\ R_e)\end{array}\right\}}{\left\{\ P\ \right\}\ s_1; s_2\ \left\{\begin{array}{ll}(\mathcal{X} = \mathit{normal} & \Rightarrow\ R_n)\ \wedge \\ (\mathcal{X} = \mathit{break} & \Rightarrow\ R_b)\ \wedge \\ (\mathcal{X} = \mathit{exc} & \Rightarrow\ R_e)\end{array}\right\}}$$

We use this notation in each proof of theorem 1.

A.2 Proof of lemmas for divide

A.2.1 Proof of lemma 1

This proof is done by induction on ∇_S and case analysis over the exception table et'. Due to et' is returned by the translation ∇_S, it only can be: either $et' = et + et_a$ where et_a is a new exception table created by ∇_S or $et' = \mathit{divide}(et, r, l'_a, l'_b)$ where et_i is stored in f and $l_a \leq l'_a < l'_b \leq l_b$. This is because ∇_S only adds new lines to the exception table taken as parameter (et) in the translation of try-catch and try-finally and ∇_S only divides the table in the translation of break.

Case $\mathbf{et' = et + et_a}$

The exception table et_a is produced by ∇_S ($\{P_n\}\ s\ \{Q\},\ m, f, et$). The generated lines contains labels between l_a and l_b. So et_a only contains exception lines between l_a and l_b. We look for the exception label between l_s and l_e where $l_b < l_s < l_e \leq l_{end}$. So, this exception cannot be defined in the exception table et_a. Then if we look for the target label in $et'[l_s, l_e, T]$ is equivalent to look for in $et[l_s, l_e, T]$.

Due to $\forall\ l_s, l_e : l_b < l_s < l_e \leq l_{end}$ we know that $l_{start} \leq l_s < l_e \leq l_{end}$. Then looking for the target label between l_s and l_e ($et[l_s, l_e, T]$) is equivalent to looking for the target label between l_{start} and l_{end} ($et[l_{start}, l_{end}, T]$). Then $et[l_s, l_e, T] = et[l_{start}, l_{end}, T]$. So, this proves that $et'[l_s, l_e, T] = et[l_{start}, l_{end}, T]$ $\square$

Case $\mathbf{et' = divide(et, r, l'_a, l'_b)}$

In this proof we use the definition of divide presented in section 6.8.1 on page 37.

There are two possible results after the execution of $\mathit{divide}(et, r, l'_a, l'_b)$ either divide does not divide the line in $et[l_{start}, l_{end}, T]$ (let call $[l_{start'}, l_{end'}, l_t, T]$ with $l_{start'} \leq l_{start}$ and $l_{end'} \leq l_{end}$) or it does.

If $\mathit{divide}(et, r, l'_a, l'_b)$ does not divide the line $[l_{start'}, l_{end'}, l_t, T]$, then using a similar reasoning to the above case, we can conclude that $et'[l_s, l_e, T] = et[l_{start}, l_{end}, T]$.

If $\mathit{divide}(et, r, l'_a, l'_b)$ divides the line $[l_{start'}, l_{end'}, l_t, T]$ then it will be divided as $[l_{start'}, l_{a'}, l_t, T]$ + $[l_{b'}, l_{end'}, l_t, T]$. We are looking for the target label between l_s and l_e. But we know $l_b < l_s < l_e \leq l_{end}$ and $l_a \leq l'_a < l'_b \leq l_b$. Then $et'[l_s, l_e, T]$ will return the target label in the divided line $[l_{b'}, l_{end'}, l_t, T]$ which target label is the same than $et[l_{start}, l_{end}, T]$. So we can conclude $et[l_{start}, l_{end}, T] = et'[l_s, l_e, T]$. $\square$

A.2.2 Proof of lemma 2

The proof is by induction on et_1.

Base case: $et_1 = [\ ]$
By definition of divide, $divide([\ ], r, l_s, l_e) = [r]$. Then the lemma holds.
Inductive Case: $divide(r_i + et'_1, r, l_s, l_e)$
If $r \not\subseteq r_i$ then divide either divides r_i or not depending on whether $(r.from > r_i.to \ \wedge \ r.to < r_i.to)$ holds or not. But it does not update r. Then by induction hypothesis we get $divide(r_i + et'_1, r, l_s, l_e)[l_s, l_e, T] = r[T]$.
If $r \subseteq r_i$ then divide returns $r_i + et'_1$. Then $r_i + et'_1[l_s, l_e, T] = r[T]$ because $r \subseteq r_i$ and divide does not change r_i.
□

A.3 Compositional Statement

The translation of compositional statement was presented in section 6.5.3 on page 31.
We have to prove:

$$
\begin{array}{l}
\vdash \{P_n\}\ s_1; s_2\ \{R\} \quad \wedge \\
[(I_{l_a} ... I_{l_b}),\ et_2] = \nabla_S \left(\{P_n\}\ s_1; s_2\ \{R\},\ m \left[\begin{array}{ll} start & \rightarrow l_a \\ next & \rightarrow l_{b+1} \end{array} \right], f, et \right) \ \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (R_e \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \ \Rightarrow \ E_{et[l_a, l_b, T]}) \ \wedge \\
\big(R_n \ \Rightarrow \ E_{m[next]}\big) \ \wedge \\
\neg(R_b \ \Rightarrow \ false) \Rightarrow \\
\left(\begin{array}{ll}
f \neq \emptyset \ \Rightarrow \ \forall i \ \in \ 1..k : & \left(\begin{array}{l}
(\vdash \{A^i\}\ s_i\ \{B^i\} \quad \wedge \quad (B^i_b \Rightarrow A^{i+1}_b) \ \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (B^i_e \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \ \Rightarrow \ E_{et_i[T]}\ ) \ \wedge \\
(Q_b \Rightarrow \ A^1_b) \ \wedge \ (B^k_b \ \Rightarrow \ E_{m[break]}) \ \wedge \\
(\forall\ et_i = r^1_i + r^2_i + ... + r^m_i : (r^1_i \cap r^2_i \cap ... \cap r^m_i = \emptyset) \ \wedge (et_i \subseteq et))
\end{array} \right) \\
f = \emptyset \ \Rightarrow \ (Q_b \ \Rightarrow \ E_{m[break]}) &
\end{array} \right) \\
\Rightarrow \\
\forall\ l \ \in \ l_a\ ...\ l_b : \vdash \{E_l\}\ I_l
\end{array}
$$

Let $m' : Event \rightarrow Label$ be a mapping function defined as:

$$m' = m \left[\ \text{next} \ \ \rightarrow l_b \ \right]$$

case: $\{\mathbf{P}\}\ \mathbf{s_1}\ \{\mathbf{Q'}\}$
By the first induction hypothesis, we get:

$$
\begin{array}{l}
\vdash \{P_n\}\ s_1\ \{Q'\} \quad \wedge \\
[(I_{l_a} ... I_{l_{a\_end}}),\ et_1] = \nabla_S(\{P_n\}\ s_1\ \{Q'\},\ m', f, et\ ) \ \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{et[l_a, l_{a\_end}, T]}) \ \wedge \\
(Q_n \ \Rightarrow\ E_{l_b}) \ \wedge \\
\neg(R_b \ \Rightarrow\ false) \Rightarrow \\
\left(
\begin{array}{l}
f \neq \emptyset \ \Rightarrow\ \forall i\ \in\ 1..k :
\left(
\begin{array}{l}
(\vdash \{A^i\}\ s_i\ \{B^i\} \quad \wedge\ \ (B_b^i \Rightarrow A_b^{i+1}) \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (B_e^i\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{et_i[T]}\ ) \wedge \\
(Q_b \Rightarrow A_b^1) \ \wedge\ (B_b^k \ \Rightarrow\ E_{m'[break]}) \wedge \\
(\forall\ et_i = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset) \ \wedge (et_i \subseteq et))
\end{array}
\right) \\
f = \emptyset \ \Rightarrow\ (Q_b \ \Rightarrow\ E_{m[break]})
\end{array}
\right) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_{a\_end} : \vdash \{E_l\}\ I_l \\
where \\
Q' \ \equiv
\left\{
\begin{array}{ll}
(\mathcal{X} = normal & \Rightarrow\ Q_n) \wedge \\
(\mathcal{X} = break & \Rightarrow\ R_b) \wedge \\
(\mathcal{X} = exc & \Rightarrow\ R_e)
\end{array}
\right\}
\end{array}
$$

We have:

$$\vdash \{P_n\}\ s_1\ \{Q'\}$$

Due to

- $m[break] = m'[break]$,

we also know by theorem hypothesis that:

$$
\begin{array}{l}
\neg(R_b \ \Rightarrow\ false) \Rightarrow \\
\left(
\begin{array}{l}
f \neq \emptyset \ \Rightarrow\ \forall i\ \in\ 1..k :
\left(
\begin{array}{l}
(\vdash \{A^i\}\ s_i\ \{B^i\} \quad \wedge\ \ (B_b^i \Rightarrow A_b^{i+1}) \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (B_e^i\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{et_i[T]}\ ) \wedge \\
(Q_b \Rightarrow A_b^1) \ \wedge\ (B_b^k \ \Rightarrow\ E_{m'[break]}) \wedge \\
(\forall\ et_i = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset) \ \wedge (et_i \subseteq et))
\end{array}
\right) \\
f = \emptyset \ \Rightarrow\ (Q_b \ \Rightarrow\ E_{m[break]})
\end{array}
\right)
\end{array}
$$

To be able to apply the first induction hypothesis we have to show that $Q_n \ \Rightarrow\ E_{l_b}$. The implication is true because $E_{l_b} \ \equiv\ Q_n$. Furthermore, we have to prove:

$$
\begin{array}{l}
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{et[l_a, l_{a\_end}, T]})
\end{array}
$$

We can prove this implication using the theorem hypothesis and the fact that $l_{a\_end} < l_b$ and we get:

$$\forall\ l\ \in\ l_a\ ...\ l_{a\_end} : \vdash \{E_l\}\ I_l$$

Case: $\{\mathbf{Q_n}\}\ \mathbf{s_2}\ \{\mathbf{R}\}$

Now we have to prove the translation of the second statement (s_2). Most of the proof is similar to the above proof. But the interesting part is the following:

$$\begin{array}{l}(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\ \qquad (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \Rightarrow\ E_{et_1[l_b, l_{b\_end}, T]})\end{array} \tag{1}$$

where et_1 is:

$$[b_1, et_1] = \nabla_S \left(T_{S_1},\ m \left[\ \textsf{next}\ \ \rightarrow l_b\ \right],\ f, et\right)$$

From the hypothesis we know:

$$\begin{array}{l}(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\ \qquad (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \Rightarrow\ E_{et[l_b, l_{b\_end}, T]})\end{array} \tag{2}$$

Applying lemma 1 on page 37 we prove 2. So, we get:

$$\forall\ l\ \in\ l_b\ ...\ l_{b\_end} : \vdash \{E_l\}\ I_l$$

Finally, we join both result and we get:

$$\forall\ l\ \in\ l_a\ ...\ l_{b\_end} : \vdash \{E_l\}\ I_l$$

□

A.4 Break statement

The translation of break statement was presented in section 6.8.1 on page 36.

We have to prove:

$$\begin{array}{l}
\vdash \{P_n\}\ break\ \{Q\}\quad \wedge \\
[(I_{l_a} ... I_{l_b}),\ et_k] = \nabla_S \left(\{P_n\}\ break\ \{Q\},\ m \begin{bmatrix} start & \rightarrow l_a \\ next & \rightarrow l_{b+1} \end{bmatrix}, f, et'_0 \right)\ \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (false\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \Rightarrow\ E_{et'_0[l_a, l_b, T]})\ \wedge \\
\big(false\ \Rightarrow\ E_{m[next]}\big)\ \wedge \\
\neg(P_n\ \Rightarrow\ false) \Rightarrow \\
\left(\begin{array}{l}
f \neq \emptyset\ \Rightarrow\ \forall i\ \in\ 1..k : \left(\begin{array}{l}
(\vdash \{A^i\}\ s_i\ \{B^i\}\quad \wedge\ \ (B^i_b \Rightarrow A^{i+1}_b)\ \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (B^i_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \Rightarrow\ E_{et_i[T]}\ )\ \wedge \\
(P_n \Rightarrow\ A^1_b)\ \wedge\ (B^k_b\ \Rightarrow\ E_{m[break]})\ \wedge \\
(\forall\ et_i = r^1_i + r^2_i + ... + r^m_i : (r^1_i \cap r^2_i \cap ... \cap r^m_i = \emptyset)\ \wedge (et_i \subseteq et))
\end{array} \right) \\
f = \emptyset\ \Rightarrow\ (P_n\ \Rightarrow\ E_{m[break]})
\end{array} \right) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_b : \vdash \{E_l\}\ I_l \\
where \\
Q\ \equiv\ \left\{ \begin{array}{lll}
(\mathcal{X} = normal & \Rightarrow & false)\ \wedge \\
(\mathcal{X} = break & \Rightarrow & P_n)\ \wedge \\
(\mathcal{X} = exc & \Rightarrow & false)
\end{array} \right\}
\end{array}$$

We have to prove the theorem for all f_i. We make the proof for:

$$[b_i, et'_i] = \nabla_S \left(t_i,\ m_i \begin{bmatrix} start & \to l_{a_i} \\ next & \to l_{a_{i+1}} \end{bmatrix},\ f_{i+1} + ... + f_k,\ divide(et'_{i-1}, et_i, l_{a_i}, l_{a_i+1}) \right)$$

and the remaining proofs are similar.

Let $et = divide(et'_{i-1}, et_i, l_{a_i}, l_{a_i+1})$. We have to prove:

$$\begin{array}{l}
\vdash \{A^i\}\ s_i\ \{B^i\} \quad \wedge \\
[I_{l_{a_i}} \ ...\ I_{l_{a_{i+1}}}, et'_i] = \nabla_S \left(\{A^i\}\ s_i\ \{B^i\},\ m_i \begin{bmatrix} start & \to l_{a_i} \\ next & \to l_{a_{i+1}} \end{bmatrix},\ f_{i+1} + ...f_k,\ et \right) \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (B^i_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{et[l_{a_i}, l_{a_{i+1}}, T]}) \ \wedge \\
(B^i_n \ \Rightarrow\ E_{m_i[next]}) \wedge \\
\neg(B^i_b \ \Rightarrow\ false) \Rightarrow \\
\left(\begin{array}{ll}
f \neq \emptyset \ \Rightarrow\ \forall j\ \in\ i+1..k : & \left(\begin{array}{l}
(\vdash \{A^j\}\ s_j\ \{B^j\} \quad \wedge\ (B^j_b \Rightarrow A^{j+1}_b) \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (B^j_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{et_j[T]}\ ) \wedge \\
(B^i_b \Rightarrow\ A^{i+1}_b)\ \wedge\ (B^k_b\ \Rightarrow\ E_{m_i[break]}) \wedge \\
(\forall\ et_i = r^1_i + r^2_i + ... + r^m_i : (r^1_i \cap r^2_i \cap ... \cap r^m_i = \emptyset)\ \wedge (et_i \subseteq et))
\end{array} \right) \\
f = \emptyset \ \Rightarrow\ (P_n \ \Rightarrow\ E_{m[break]}) &
\end{array} \right) \\
\Rightarrow \\
\forall\ l\ \in\ l_{a_i}\ ...\ l_{a_{i+1}} : \vdash \{E_l\}\ I_l
\end{array}$$

We can prove:

$$\begin{array}{l}
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (B^i_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{et[l_{a_i}, l_{a_{i+1}}, T]})
\end{array}$$

by using the hypothesis (case $f \neq \emptyset$):

$$\begin{array}{l}
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (B^i_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{et_i[T]}\ )
\end{array}$$

and lemma 2 that says $et[l_{a_i}, l_{a_{i+1}}, T] = et_i[T]$.

Due to $m[next] = l_{a_{i+1}}$ then $B^i_n \ \Rightarrow\ E_{m[next]}$ is equivalent to prove $B^i_n \ \Rightarrow\ E_{l_{a_{i+1}}}$. $E_{l_{a_{i+1}}} \equiv A^{i+1}$, then we have to prove $B^i_n \ \Rightarrow\ A^{i+1}$. This is true because by the hypothesis we know that $B^i_b \ \Rightarrow\ A^{i+1}$ and also we know that $B^i_b \ \Rightarrow\ B^i_n$.

We can prove the finally proof $f_i...f_k$

$$\begin{array}{l}
\neg(B^i_b \ \Rightarrow\ false) \Rightarrow \\
\left(\begin{array}{ll}
f \neq \emptyset \ \Rightarrow\ \forall j\ \in\ i+1..k : & \left(\begin{array}{l}
(\vdash \{A^j\}\ s_j\ \{B^j\} \quad \wedge\ (B^j_b \Rightarrow A^{j+1}_b) \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (B^j_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{et_j[T]}\ ) \wedge \\
(B^i_b \Rightarrow\ A^{i+1}_b)\ \wedge\ (B^k_b\ \Rightarrow\ E_{m[break]}) \wedge \\
(\forall\ et_i = r^1_i + r^2_i + ... + r^m_i : (r^1_i \cap r^2_i \cap ... \cap r^m_i = \emptyset)\ \wedge (et_i \subseteq et))
\end{array} \right) \\
f = \emptyset \ \Rightarrow\ (P_n \ \Rightarrow\ E_{m[break]}) &
\end{array} \right)
\end{array}$$

by hypothesis. In the translation, we change the list f by $f_i...f_k$. But from the hypothesis we know that it holds for $f_1...f_k$.

Now we can apply the hypothesis and get $\forall\ l\ \in\ l_{a_i}\ ...\ l_{ai+1} : \vdash \{E_l\}\ I_l$

Finally, we have to prove that $P_n\ \Rightarrow\ A_{i_b}$ and for all i: $B_b^i \Rightarrow\ A_b^{i+1}$ and $B_b^k\ \Rightarrow\ E_{m[break]}$. This holds by the hypothesis.

□

A.5 Translation for while considering break statement and exceptions

The translation of while statement was presented in section 6.8.2 on page 38.

We have to prove:

$$
\begin{array}{l}
\vdash \{I_n\}\ \ while\ (e)\ s_1\ \ \{\ Q\ \}\ \ \ \wedge \\
[(I_{l_a}...I_{l_d}),\ et_1] = \nabla_S\left(\{I_n\}\ while\ (e)\ s_1\ \{Q\},\ m\left[\begin{array}{ll} start & \rightarrow l_a \\ next & \rightarrow l_{d+1} \end{array}\right], f, et\right)\ \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \ \Rightarrow\ \ E_{et[l_a,l_b,T]})\ \ \wedge \\
\big(((I_n\ \ \wedge \neg e)\ \vee\ Q_b\ )\ \ \Rightarrow\ \ E_{m[next]}\big)\ \wedge \\
\neg(false\ \ \Rightarrow\ \ false) \Rightarrow \\
\left(\begin{array}{ll}
f \neq \emptyset\ \Rightarrow\ \forall i\ \in\ 1..k : & \left(\begin{array}{l}
(\vdash \{A^i\}\ s_i\ \{B^i\}\quad \wedge\ \ (B_b^i \Rightarrow A_b^{i+1})\ \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (B_e^i\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \ \Rightarrow\ \ E_{et_i[T]}\ )\ \wedge \\
(false \Rightarrow\ A_b^1)\ \wedge\ (B_b^k\ \ \Rightarrow\ E_{m[break]})\ \wedge \\
(\forall\ et_i = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset)\ \wedge (et_i \subseteq et))
\end{array}\right) \\
f = \emptyset\ \Rightarrow\ (false\ \ \Rightarrow\ \ E_{m[break]}) &
\end{array}\right) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_d : \vdash \{E_l\}\ I_l \\
where \\
Q\ \equiv \left\{\begin{array}{lll}
(\mathcal{X} = normal & \Rightarrow & (\ (I_n \wedge \neg e) \vee\ Q_b\ ))\ \wedge \\
(\mathcal{X} = break & \Rightarrow & false)\ \wedge \\
(\mathcal{X} = exc & \Rightarrow & R_e)
\end{array}\right\}
\end{array}
$$

Let $m' : Event\ \rightarrow\ Label$ be a mapping function defined as:

$$
m' = m\left[\begin{array}{ll} \mathsf{start} & \rightarrow l_b \\ \mathsf{next} & \rightarrow l_c \\ \mathsf{break} & \rightarrow m[next] \end{array}\right]
$$

and $f' = \emptyset$

By the induction hypothesis, we get:

$$
\begin{array}{l}
\vdash \{e \ \wedge \ I_n\} \ s_1 \ \{\ Q' \ \} \quad \wedge \\
[(I_{l_b} ... I_{l_{b\_end}}), \ et_1] = \nabla_S(\{e \ \wedge \ I_n\} \ s_1 \ \{Q'\}, \ m', f', et \ ) \ \wedge \\
(\forall \ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (R_e \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \ \ \Rightarrow \ \ E_{et[l_a, l_b, T]}) \ \ \wedge \\
\big(((I_n \ \ \wedge \neg e) \ \vee \ Q_b \ ) \ \ \Rightarrow \ \ E_{m[next]}\big) \ \wedge \\
\neg(Q_b \ \Rightarrow \ false) \Rightarrow \\
\left(
\begin{array}{l}
f' \neq \emptyset \ \Rightarrow \ \forall i \ \in \ 1..k : \left(
\begin{array}{l}
(\vdash \{A^i\} \ s_i \ \{B^i\} \quad \wedge \ \ (B^i_b \Rightarrow A^{i+1}_b) \wedge \\
(\forall \ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (B^i_e \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \ \ \Rightarrow \ \ E_{et_i[T]} \ ) \wedge \\
(I_n \Rightarrow \ A^1_b) \ \wedge \ (B^k_b \ \Rightarrow \ E_{m'[break]}) \wedge \\
(\forall \ et_i = r^1_i + r^2_i + ... + r^m_i : (r^1_i \cap r^2_i \cap ... \cap r^m_i = \emptyset) \ \wedge (et_i \subseteq et))
\end{array}
\right) \\
f' = \emptyset \ \Rightarrow \ (Q_b \ \Rightarrow \ E_{m'[break]})
\end{array}
\right) \\
\Rightarrow \\
\forall \ l \ \in \ l_b \ ... \ l_{b\_end} : \vdash \{E_l\} \ I_l \\
where \\
Q \ \equiv \left\{
\begin{array}{lll}
(\mathcal{X} = normal & \Rightarrow & I_n) \wedge \\
(\mathcal{X} = break & \Rightarrow & Q_b) \wedge \\
(\mathcal{X} = exc & \Rightarrow & R_e)
\end{array}
\right\}
\end{array}
$$

We have:

$$\vdash \{e \ \wedge \ I_n\} \ s_1 \{Q'\}$$

Due to the exception table et is the same than the theorem hypothesis, we know:

$$
\begin{array}{l}
(\forall \ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (R_e \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \ \ \Rightarrow \ \ E_{et[l_a, l_b, T]})
\end{array}
$$

To be able to apply the induction hypothesis we have to show that $Q_b \ \Rightarrow \ E_{m'[break]}$ (because $f' = \emptyset$) and $I_n \ \Rightarrow \ E_{l_c}$.

In the first implication we have $Q_b \ \Rightarrow \ E_{m'[break]} \equiv E_{m[next]}$. From the theorem hypothesis we have $((I_n \ \ \wedge \neg e) \ \vee \ Q_b \ ) \ \ \Rightarrow \ \ E_{m[next]}$. Then $Q_b \ \Rightarrow \ E_{m'[break]}$ holds because $Q_b \Rightarrow ((I_n \ \ \wedge \neg e) \ \vee \ Q_b \ ) \ \Rightarrow \ E_{m[next]}$.

$I_n \ \Rightarrow \ E_{l_c}$ because $E_{l_c} \ \equiv \ I_n$.

Now, we can apply the induction hypothesis and we get:

$$\forall \ l \ \in \ l_b \ ... \ l_{b\_end} : \vdash \{E_l\} \ I_l$$

The last proof we have to prove is:

$$\forall \ l \ \in \ \{l_a, l_c, l_d\} : \vdash \{E_l\} \ I_l$$

This is simple, we have to apply the definition of wp^1_p (figure 5 on page 23).

□

A.6 Try catch translation

The translation of try-catch statement was presented in section 6.9.1 on page 38.

Let S be the following postcondition:

$$
\begin{array}{l}
S \equiv \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & Q_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & Q_b) \wedge \\ (\mathcal{X} = exc & \Rightarrow & S_e) \end{array} \right\} \\
where \\
S_e \equiv \left\{ \begin{array}{l} R_e \vee \\ (Q_e \ \wedge \ \tau(excV) \not\preceq T) \end{array} \right\} \\
and \\
U \equiv \left\{ \begin{array}{lll} (\mathcal{X} = normal & \Rightarrow & Q_n) \wedge \\ (\mathcal{X} = break & \Rightarrow & Q_b) \wedge \\ (\mathcal{X} = exc & \Rightarrow & U_e) \end{array} \right\} \\
where \\
U_e \equiv \left\{ \left(\begin{array}{l} (Q_e \ \wedge \ \tau(excV) \not\preceq T) \vee \\ (Q'_e \ \wedge \ \tau(excV) \preceq T) \end{array} \right) \right\}
\end{array}
$$

We have to prove:

$$
\begin{array}{l}
\vdash \{P_n\} \ try \ s_1 \ catch \ (T \ e) \ s_2 \ \{ \ S \ \} \ \ \wedge \\
[(I_{l_a} ... I_{l_d}), \ et_2] = \nabla_S \left(\{P_n\} \ try \ s_1 \ catch \ (T \ e) \ s_2 \ \{ \ S \ \}, \ m \left[\begin{array}{ll} start & \rightarrow l_a \\ next & \rightarrow l_{d+1} \end{array} \right], f, et \right) \ \wedge \\
(\forall \ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (S_e \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \ \ \Rightarrow \ \ E_{et[l_a, l_b, T]}) \ \ \wedge \\
\big(Q_n \ \ \Rightarrow \ \ E_{m[next]}\big) \ \wedge \\
\neg(Q_b \ \Rightarrow \ false) \Rightarrow \\
\left(\begin{array}{ll}
f \neq \emptyset \ \Rightarrow \ \forall i \ \in \ 1..k : & \left(\begin{array}{l}
(\vdash \{A^i\} \ s_i \ \{B^i\} \quad \wedge \ \ (B^i_b \Rightarrow A^{i+1}_b) \ \wedge \\
(\forall \ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (B^i_e \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \ \ \Rightarrow \ \ E_{et_i[T]} \ ) \ \wedge \\
(Q_n \Rightarrow \ A^1_b) \ \wedge \ (B^k_b \ \Rightarrow \ E_{m[break]}) \ \wedge \\
(\forall \ et_i = r^1_i + r^2_i + ... + r^m_i : (r^1_i \cap r^2_i \cap ... \cap r^m_i = \emptyset) \ \wedge (et_i \subseteq et))
\end{array} \right) \\
f = \emptyset \ \Rightarrow \ (Q_b \ \Rightarrow \ E_{m[break]}) &
\end{array} \right) \\
\Rightarrow \\
\forall \ l \ \in \ l_a \ ... \ l_d : \vdash \{E_l\} \ I_l
\end{array}
$$

Let m' be a mapping function and et' be an exception table defined as:

$$
\begin{array}{l}
m' = m \left[\ \mathsf{next} \ \ \rightarrow l_b \ \right] \\
et' = et + [l_a, l_b, l_c, T]
\end{array}
$$

Case $\{\mathbf{P_n}\} \ \mathbf{s_1} \ \{\mathbf{U}\}$

By the induction hypothesis, we get:

$$
\begin{array}{l}
\vdash \{P_n\}\ \ s_1\ \ \{\ U\ \}\ \ \wedge \\
[(I_{l_a}...I_{l_{a\_end}}),\ et_1] = \nabla_S\,(\{P_n\}\ \ s_1\ \ \{\ U\ \},\ m', f, et'\ )\ \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (U_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \ \Rightarrow\ \ E_{et'[l_a,l_b,T]})\ \ \wedge \\
\big(Q_n\ \ \Rightarrow\ \ E_{m'[next]}\big)\ \wedge \\
\neg(Q_b\ \Rightarrow\ false) \Rightarrow \\
\left(\begin{array}{l}
f \neq \emptyset\ \Rightarrow\ \forall i\ \in\ 1..k : \left(\begin{array}{l}
(\vdash \{A^i\}\ s_i\ \{B^i\} \quad \wedge\ \ (B_b^i \Rightarrow A_b^{i+1})\ \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (B_e^i\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \ \Rightarrow\ \ E_{et_i[T]}\ )\ \wedge \\
(Q_n \Rightarrow\ A_b^1)\ \wedge\ (B_b^k\ \Rightarrow\ E_{m'[break]})\ \wedge \\
(\forall\ et_i = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset)\ \wedge (et_i \subseteq et))
\end{array}\right) \\
f = \emptyset\ \Rightarrow\ (Q_b\ \Rightarrow\ E_{m'[break]})
\end{array}\right) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_{a\_end} : \vdash \{E_l\}\ I_l
\end{array}
$$

We have:

$$\vdash \{P_n\}\ s_1\ \{Q\}$$

Due to

- $m[break] = m'[break]$, and
- $E_{m'[next]} = E_{l_b} = Q_n\ \ then\ \ Q_n \Rightarrow Q_n$.

the only condition we have to prove to be able to apply the induction hypothesis is:

$$
\begin{array}{l}
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (U_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \ \Rightarrow\ \ E_{et'[l_a,l_b,T]})
\end{array}
$$

U_e is defined as:

$$U_e \equiv \left\{\ \left(\begin{array}{l} (Q_e\ \wedge\ \tau(excV) \npreceq T) \vee \\ (Q'_e\ \wedge\ \tau(excV) \preceq T) \end{array}\right)\ \right\}$$

If s_1 throws an exception and the type of the exception is not less than T then $Q_e \wedge \tau(excV) \npreceq T$ holds. Then $Q_e\ \wedge\ \tau(excV) \npreceq T \Rightarrow S_e$. And we know from the hypothesis:

$$
\begin{array}{l}
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (S_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \ \Rightarrow\ \ E_{et[l_a,l_b,T]})
\end{array}
$$

Then

$$
\begin{array}{l}
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (U_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \ \Rightarrow\ \ E_{et'[l_a,l_b,T]})
\end{array}
$$

holds.

If s_1 throws an exception and the type of the exception is less than T then $Q'_e\ \wedge\ \tau(excV) \preceq T$ holds. et' is defined as $et + [l_a, l_b, l_c, T]$. Then $E_{et'[l_a,l_b,T]} = l_c$. We have to prove that:

$$Q'_e\ \wedge\ excV \neq null\ \wedge\ \tau(excV) \preceq T\ \wedge\ s(0) = excV \Rightarrow E_{l_c} \tag{3}$$

$$
\begin{array}{l}
(\forall\ T : Type - \{T\} : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (U_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \ \Rightarrow\ \ E_{et'[l_a,l_b,T]}
\end{array} \tag{4}
$$

(4) holds from the hypothesis. (3) holds because $E_{l_c} \equiv shift(Q'_e) \ \wedge \ excV \neq null \ \wedge \ \tau(excV) \preceq T \ \wedge \ s(0) = excV$ and $shift(Q'_e) = Q'_e$ due to Q'_e does not refer to the stack.

Then, we can apply the induction hypothesis and we get:

$$\forall \ l \ \in \ l_a \ ... \ l_{a\_end} : \vdash \{E_l\} \ I_l$$

Applying a similar reasoning we can apply the second induction hypothesis and we get:

$$\forall \ l \ \in \ \{l_d, ...l_{d\_end}\} : \vdash \{E_l\} \ I_l$$

We can prove $\{ \ Q_n \ \} \ l_b$: goto $m[next]$ by using the hypothesis. Finally we can prove $\{ \ E_{l_c} \ \}$ pop e using wp^1_p definition.

Joining the proofs, we get

$$\forall \ l \ \in \ l_a \ ... \ l_d : \vdash \{E_l\} \ I_l$$

□

A.7 Throw Translation

The translation of throw statement was presented in section 6.9.2 on page 39.

We have to prove:

$$\begin{array}{l}
\vdash \{P_n[e/excV]\} \quad throw \ e \quad \{ \ Q \ \} \quad \wedge \\
[(I_{l_a}...I_{l_b}), \ et_1] = \nabla_S \left(\{P_n[e/excV]\} \quad throw \ e \quad \{ \ Q \ \}, \ m \begin{bmatrix} start & \rightarrow l_a \\ next & \rightarrow l_{b+1} \end{bmatrix}, f, et \right) \ \wedge \\
(\forall \ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (Q_e \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \quad \Rightarrow \quad E_{et[l_a, l_b, T]}) \quad \wedge \\
(false \quad \Rightarrow \quad E_{m[next]}) \ \wedge \\
\Rightarrow \\
\forall \ l \ \in \ l_a \ ... \ l_b : \vdash \{E_l\} \ I_l \\
where \\
Q \ \equiv \left\{ \begin{array}{ll} (\mathcal{X} = normal & \Rightarrow \ false) \ \wedge \\ (\mathcal{X} = break & \Rightarrow \ false) \ \wedge \\ (\mathcal{X} = exc & \Rightarrow \ P_n) \end{array} \right\}
\end{array}$$

$\vdash \{E_{l_a}\} \ I_{l_a}$ holds because of the definition of wp^1_p. $\vdash \{E_{l_b}\} \ I_{l_b}$ holds by theorem hypothesis and also the definition of wp^1_p.

□

A.8 Translation of Finally statements for Java

The translation of finally statement was presented in section 6.9.3 on page 40.

We have to prove:

$$
\begin{array}{l}
\vdash \{P_n\}\ \ try\ s_1\ finally\ s_2\ \ \{\ V\ \}\quad \wedge \\
\left[(I_{l_a}...I_{l_g}),\ et_3\right] = \nabla_S \left(\{P_n\}\ \ try\ s_1\ finally\ s_2\ \ \{\ V\ \},\ m \left[\begin{array}{ll} start & \rightarrow l_a \\ next & \rightarrow l_{g+1} \end{array} \right], f, et \right)\ \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (Q'_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \ \Rightarrow\ \ E_{et[l_a,l_b,T]})\ \ \wedge \\
\left(Q'_n\ \ \Rightarrow\ \ E_{m[next]}\right)\ \wedge \\
\neg((Q'_b\ \vee\ R_b)\ \Rightarrow\ false) \Rightarrow \\
\left(\begin{array}{l}
f \neq \emptyset\ \Rightarrow\ \forall i\ \in\ 1..k : \left(\begin{array}{l}
(\vdash \{A^i\}\ s_i\ \{B^i\}\quad \wedge\ \ (B^i_b \Rightarrow A^{i+1}_b)\ \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (B^i_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \ \Rightarrow\ \ E_{et_i[T]}\ )\ \wedge \\
((Q'_b\ \vee\ R_b) \Rightarrow\ A^1_b)\ \wedge\ (B^k_b\ \ \Rightarrow\ E_{m[break]})\ \wedge \\
(\forall\ et_i = r^1_i + r^2_i + ... + r^m_i : (r^1_i \cap r^2_i \cap ... \cap r^m_i = \emptyset)\ \wedge (et_i \subseteq et))
\end{array} \right) \\
f = \emptyset\ \Rightarrow\ ((Q'_b\ \vee\ R_b)\ \Rightarrow\ E_{m[break]})
\end{array} \right) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_g : \vdash \{E_l\}\ I_l
\end{array}
$$

Let $m' : Event\ \rightarrow\ Label$ be a mapping function defined as:

$$m' = m\left[\ \mathsf{next}\ \ \rightarrow l_b\ \right]$$

Let f' be a list of $Finally$ defined as:

$$f' = [\{T\}\ s_2\ \{U\}\ ,\ getExceptions(l_a, l_b, et)] + f$$

Let et' be an exception table defined as:

$$et' = et + [l_a, l_b, l_d, any]$$

By the induction hypothesis, we get:

$$
\begin{array}{l}
\vdash \{P_n\}\ \ s_1\ \ \{\ S\ \}\quad \wedge \\
[(I_{l_a}...I_{l_{a\_end}}),\ et_1] = \nabla_S\left(\{P_n\}\ \ s_1\ \ \{\ S\ \},\ m', f', et'\ \right)\ \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (Q_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \ \Rightarrow\ \ E_{et'[l_a,l_b,T]})\ \ \wedge \\
(Q_n\ \ \Rightarrow\ \ E_{l_b})\ \wedge \\
\neg(Q_b\ \Rightarrow\ false) \Rightarrow \\
\left(\begin{array}{l}
f \neq \emptyset\ \Rightarrow\ \forall i\ \in\ 1..k : \left(\begin{array}{l}
(\vdash \{A^i\}\ s_i\ \{B^i\}\quad \wedge\ \ (B^i_b \Rightarrow A^{i+1}_b)\ \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (B^i_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \ \Rightarrow\ \ E_{et_i[T]}\ )\ \wedge \\
(Q_b\ \ \Rightarrow\ A^1_b)\ \wedge\ (B^k_b\ \ \Rightarrow\ E_{m'[break]})\ \wedge \\
(\forall\ et_i = r^1_i + r^2_i + ... + r^m_i : (r^1_i \cap r^2_i \cap ... \cap r^m_i = \emptyset)\ \wedge (et_i \subseteq et))
\end{array} \right) \\
f = \emptyset\ \Rightarrow\ Q_b\ \Rightarrow\ E_{m'[break]})
\end{array} \right) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_{a\_end} : \vdash \{E_l\}\ I_l
\end{array}
$$

We have:

$$\vdash \{P_n\}\ s_1\ \{S\}$$

Due to $et' = et + [l_a, l_b, l_d, any]$, then $et'[l_a, l_b, T] = l_d, \forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any))$. Then we have to prove that:

$$\begin{pmatrix} Q_e\ \wedge\ \tau(excV) \preceq T\ \wedge \\ excV \neq null\ \wedge\ s(0) = excV) \end{pmatrix} \Rightarrow \begin{pmatrix} shift(Q_e)[eTmp/excV]\ \wedge \\ \wedge\ eTmp = excV \end{pmatrix}$$

The implication holds due to Q_e does not refer to the stack and then $shift(Q_e)\ \equiv\ Q_e$.

To be able to apply the first induction hypothesis, we need to prove:

$$\neg(Q_b\ \Rightarrow\ false) \Rightarrow \left(\begin{array}{l} f \neq \emptyset\ \Rightarrow\ \forall i\ \in\ 1..k : \left(\begin{array}{l} (\vdash \{A^i\}\ s_i\ \{B^i\}\quad \wedge\ \ (B^i_b \Rightarrow A^{i+1}_b)\ \wedge \\ (\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\ \qquad (B^i_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \Rightarrow\ E_{et_i[T]}\ )\ \wedge \\ (Q_b\ \Rightarrow\ A^1_b)\ \wedge\ (B^k_b\ \Rightarrow\ E_{m'[break]})\ \wedge \\ (\forall\ et_i = r^1_i + r^2_i + ... + r^m_i : (r^1_i \cap r^2_i \cap ... \cap r^m_i = \emptyset)\ \wedge (et_i \subseteq et)) \end{array} \right) \\ f = \emptyset\ \Rightarrow\ Q_b\ \Rightarrow\ E_{m'[break]}) \end{array} \right)$$

Let et''_i be the exception table for the current finally block. $et''_i = getExceptions(l_a, l_b, et)$. Using the definition of f', we need to prove:

1. $Q_b\ \Rightarrow\ A^0_b$
2. $B^k_b\ \Rightarrow\ E_{m'[break]}$
3. $B^0_b \Rightarrow A^1_b$
4. $\begin{array}{l}(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\ \qquad (B^i_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \Rightarrow\ E_{et''_i[T]}\ )\end{array}$
5. $(\forall\ et''_i = r^1_i + r^2_i + ... + r^m_i : (r^1_i \cap r^2_i \cap ... \cap r^m_i = \emptyset)\ \wedge (et''_i \subseteq et))$

1. $Q_b\ \Rightarrow\ A^0_b$ holds due to $A^0_b\ \equiv\ Q_b$.
2. $B^k_b\ \Rightarrow\ E_{m'[break]}$ holds because B^k_b is the same than the theorem hypothesis and the implication holds in the theorem hypothesis.
3. $B^0_b \Rightarrow A^1_b$: from theorem hypothesis, we know that $(Q'_b\ \vee\ R_b) \Rightarrow\ A^1_b$. $B^0_b\ \equiv\ Q'_b$ then $Q'_b\ \Rightarrow\ A^1_b$
4. We have to prove $\begin{array}{l}(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\ \qquad (Q'_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \Rightarrow\ E_{et''_i[T]}\end{array}$

   because $B^0_b \equiv Q'_e$. We get the exceptions lines of et''_i from et. Then the implication can be proved by hypothesis.
5. This holds from the definition of *getExceptions* because it returns different exception lines.

Now, we can apply the first induction hypothesis and we get:

$$\forall\ l\ \in\ l_a\ ...\ l_{a\_end} : \vdash \{E_l\}\ I_l$$

Applying a similar reasoning than the proof for try and catch statements we get:

$$\forall\ l\ \in\ \{l_b, ...l_g\} : \vdash \{E_l\}\ I_l$$

Joining the proofs, we get

$$\forall\ l\ \in\ l_a\ ...\ l_g : \vdash \{E_l\}\ I_l$$

□

B Appendix: Logic for finally statements in C# and its translation

B.1 Finally Rule for C#

As we mentioned in section 3.5.2, the C# compiler does not allow one to write break statements inside of finally clauses. Due to this, we do not need to consider that the statement s_2 executes a break. The postcondition of s_2 for the break case, is always $\mathcal{X} = break \Rightarrow\ false$. The rule is similar to the rule presented in 3.5.2 (finally statements in Java) but s_2 never executes a break.

$$\dfrac{\begin{array}{c}\left\{\ P\ \right\}\ s_1\ \left\{\begin{array}{lll}(\mathcal{X} = normal & \Rightarrow & Q_n)\ \wedge\\ (\mathcal{X} = break & \Rightarrow & Q_b)\ \wedge\\ (\mathcal{X} = exc & \Rightarrow & Q_e)\end{array}\right\}\\[2ex] \left\{\begin{array}{c}(Q_n\ \wedge\ \mathcal{X}Tmp = normal)\ \vee\\ (Q_b\ \wedge\ \mathcal{X}Tmp = break)\ \vee\\ \left(\begin{array}{l}(Q_e[eTmp/excV]\ \wedge\\ \mathcal{X}Tmp = exc\ \wedge\\ eTmp = excV)\end{array}\right)\end{array}\right\}\ s_2\ \left\{\begin{array}{lll}(\mathcal{X} = normal & \Rightarrow & \left(\begin{array}{l}(Q'_n\ \wedge\ \mathcal{X}Tmp = normal)\ \vee\\ (Q'_b\ \wedge\ \mathcal{X}Tmp = break)\ \vee\\ (Q'_e\ \wedge\ \mathcal{X}Tmp = exc)\end{array}\right))\ \wedge\\ (\mathcal{X} = break & \Rightarrow & false)\ \wedge\\ (\mathcal{X} = exc & \Rightarrow & Q'_e)\end{array}\right\}\end{array}}{\left\{\ P\ \right\}\ try\ s_1\ finally\ s_2\ \left\{\begin{array}{lll}(\mathcal{X} = normal & \Rightarrow & Q'_n)\ \wedge\\ (\mathcal{X} = break & \Rightarrow & Q'_b)\ \wedge\\ (\mathcal{X} = exc & \Rightarrow & Q'_e)\end{array}\right\}}$$

B.2 Translation of finally statements for C#

Let S, T, U, V be

$$S \equiv \left\{\begin{array}{c}(\mathcal{X} = normal\ \ \Rightarrow\ \ Q_n)\ \ \wedge\\ (\mathcal{X} = break\ \ \Rightarrow\ \ Q_b)\ \wedge\\ (\mathcal{X} = exc\ \ \Rightarrow\ \ Q_e)\end{array}\right\}$$

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

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

$$V \equiv \left\{\begin{array}{c}(\mathcal{X} = normal\ \ \Rightarrow\ \ Q'_n\ )\ \ \wedge\\ (\mathcal{X} = break\ \ \Rightarrow\ \ Q'_b\ )\ \wedge\\ (\ \mathcal{X} = exc\ \ \Rightarrow\ \ (Q'_e\ \vee\ R_e)\ )\end{array}\right\}$$

Let m_1 be an mapping function, and et', et'' be exception tables

$$\begin{array}{l} m_1 \equiv m\left[\ \mathsf{next} \quad \rightarrow l_b\ \right] \\ et' = et + [l_a, l_b, l_d, any] \\ et'' = getExceptions(l_a, l_b, et')^3 \end{array}$$

$$[B_{S_1}, et_1] = \quad \nabla_S \left(\frac{Tree_1}{\{\ P_n\ \}\quad s_1 \quad \{\ S\ \}}, \ m_1\ , \left[\frac{Tree_2}{\{\ T\ \}\quad s_2 \quad \{\ U\ \}}\ , et'' \right] + f,\ et' \right)$$

$$[B_{S_2}, et_2] = \quad \nabla_S \left(\frac{Tree_2}{\{\ T\ \}\quad s_2 \quad \{\ U\ \}}, \ m \left[\begin{array}{ll} \mathsf{start} & \rightarrow l_b \\ \mathsf{next} & \rightarrow l_c \end{array} \right],\ f,\ et_1 \right)$$

$$b_{goto} = \quad \{Q'_n\} \qquad\qquad l_c : \mathsf{goto}\ m[next]$$

$$b_{pop} = \left\{ \begin{array}{l} shift(Q_e) \wedge \\ excV \neq null \\ \wedge\ s(0) = excV \end{array} \right\} \qquad l_d : \mathsf{pop}\ eTmp$$

$$[B'_{S_2}, et_3] = \quad \nabla_S \left(\frac{Tree_2}{\{\ T\ \}\quad s_2 \quad \{\ U\ \}}, \ m \left[\begin{array}{ll} \mathsf{start} & \rightarrow l_e \\ \mathsf{next} & \rightarrow l_f \end{array} \right],\ f,\ et_2 \right)$$

$$b_{pushv} = \quad \{\ Q'_n\ \vee\ Q'_b\ \vee\ Q'_e\ \} \qquad l_f : \mathsf{pushv}\ eTmp$$

$$b_{athrow} = \left\{ \begin{array}{l} (Q'_n\ \vee\ Q'_b\ \vee\ Q'_e) \\ \wedge\ s(0) = eTmp \end{array} \right\} \qquad l_g : \mathsf{athrow}$$

$$\nabla_S \left(\frac{\dfrac{Tree_1}{\left\{\ P_n\ \right\}\quad s_1 \quad \left\{\ S\ \right\}} \qquad \dfrac{Tree_2}{\left\{\ T\ \right\}\quad s_2 \quad \left\{\ U\ \right\}}}{\left\{\ P_n\ \right\}\quad try\ s_1\ finally\ s_2 \quad \left\{\ V\ \right\}},\ m,\ f,\ et \right) =$$

$$[\ B_{S_1} + B_{S_2} + b_{goto} + b_{pop} + B'_{S_2} + b_{pushv} + b_{athrow}\ ,\ et_3\ ]$$

Proof of theorem 1: case finally statements for C#
The proof is similar to the proof of finally statements for Java.
□

[3]getExceptions returns the exceptions of the exception table et' between l_a and l_b. It returns at most one exception line for every type.