Formalizing Proof-Transforming Compilation of Eiffel programs

Martin Nordio[1], Peter Müller[2], and Bertrand Meyer [1]

(1) ETH Zurich, Switzerland
{`Martin.Nordio, Bertrand.Meyer`}`@inf.ethz.ch`

(2) Microsoft Research, USA
`mueller@microsoft.com`

ETH Technical Report 587

February 2008

Abstract

The execution of mobile code can produce unexpected behavior, which may comprise security and correctness of a software system. Proof-Carrying Code allows one to execute mobile code in a safe way by checking a formal proof before the code is executed. However, automatic generation of proofs works only for basic safety properties such as type safety.

To apply PCC to functional correctness properties, we propose to verify the source program interactively and then to translate the proof to bytecode. This proof translation is relatively straightforward if the source and target language are similar, such as Java and Java bytecode, but poses challenges for more complex translations. In this paper, we present a proof-transforming compiler for a subset of Eiffel to the .NET CIL. In particular, we show how the non-trivial translations of multiple inheritance and Eiffel's exceptions can be handled.

Keywords: Software verification, program proofs, Proof-Carrying Code, proof-transforming compiler, Eiffel, CIL

Contents

1 Introduction

Proof-Carrying Code (PCC) [12] enables the safe execution of mobile code. Code producers develop the code together with a formal proof (a certificate) that the code has certain desirable properties. The code consumer checks the proof before executing the mobile code.

Code producers can use certifying compilers [13] to generate proofs for basic properties such as type safety automatically. However, the verification of functional correctness typically requires interactive verification, which is beyond the capabilities of certifying compilers. Alternatively, code producers can interactively verify the executable mobile code (that is, the bytecode). However, reasoning about bytecode is difficult due to its unstructured control ow and weak type system.

We propose to interactively verify the desired properties for the source program and to translate the obtained proof to bytecode. This proof translation can be performed automatically by a *proof-transforming compiler* (PTC). A PTC is similar to a certifying compiler, but takes a source program, its specification, and a source proof as input and produce the bytecode program, specification, and proof. On the consumer side, a proof checker ensures that the proof actually guarantees that the program satisfies its specification.

An important property of proof-transforming compilers is that they are not part of the trusted code base of the PCC infrastructure. If the compiler produces a wrong specification or a wrong proof for a component, the proof checker will reject the component. This approach combines the strengths of certifying compilers and interactive verification. Our proof-transforming compiler consists of two modules: (1) a specification translator that translates Eiffel contracts to CIL contracts; and (2) a proof translator that translates Eiffel proofs to CIL proofs. The specification translator takes an Eiffel contract (based on Eiffel expressions) and generates a CIL contract (based on First order logic). The proof translator takes a proof in a Hoare-style logic and generates a CIL bytecode proof.

Proof-transforming compilation can be fairly straightforward if the source and the target language are very similar. For example, PTCs have been developed from Java to bytecode [1, 3, 14]. The translation is more complex when the subset is extended with `finally` and `break` statements [11]. But the difficulty of the problem grows with the conceptual distance between the semantic models of the source and target languages. In the present work, the source language is Eiffel, whose object model and type system differ significantly from the assumptions behind CIL, the target language. In particular, Eiffel supports multiple inheritance and a specific form of exception handling. This has required, in the implementation of Eiffel for .NET (which goes through CIL code), the design of original compilation techniques. In particular [6], the compilation of each Eiffel class produces two CIL types: an interface, and an implementation class which implements it. If either the source proof or the source specification expresses properties about the type structure of the Eiffel program, the same property has to be generated for the bytecode.

The translation of these properties raises challenges illustrated by the following example involving a reflective capability: the feature *type*, which gives the type of an object.

```
1 merge (other: LINKED_LIST [G]):LINKED_LIST [G] is
        -- Merge 'other' into current structure returning a new LINKED_LIST
3    require
         is_linked_list : other.type.conforms_to (LINKED_LIST [G].type)
5        same_type: Current.type.is_equal(other.type)
     ensure
7        result_type : Result.type.is_equal(LINKED_LIST [G].type)
```

The function *merge* is defined in the class *LINKED_LIST*. The precondition of *merge* expresses that the type of *other* is a subtype of *LINKED_LIST* and the types of *Current* and *other* are equal. The postcondition expresses that the type of *Result* is equal to *LINKED_LIST*.

The compilation of the class *LINKED_LIST* produces the CIL interface *LINKED_LIST_INT* and the implementation class *LINKED_LIST_IMP*. A correct PTC has to map the type *LINKED_LIST* in the clause *is_linked_list* (line 4) to the CIL interface *LINKED_LIST_INT* because in the target model decedents of the Eiffel class *LINKED_LIST* inherit from the interface *LINKED_LIST_INT* in CIL and not from *LINKED_LIST_IMP*. To translate the postcondition, we use the implementation class *LINKED_LIST_IMP* because this property expresses that the type of *Result* is equal to

LINKED_LIST. Thus, the PTC has to map Eiffel classes to CIL interfaces or Eiffel classes to CIL classes depending of the function used to express the source property.

This example illustrates that the proof-transforming compiler cannot always treat Eiffel in the same way: while in most circumstances it will map them to CIL interfaces, in some cases (such as this one, involving reflection) it must use a CIL class.

Outline. The source language and its Hoare-style logic are introduced in Sections 2 and 3. We present the bytecode language and its logic in Section 4 and 5. Section 7 presents the specification translator. In Section 6, we define the proof transformation. Section 8 illustrates proof transformations by an example. Section 9 states a soundness theorem. Related work is discussed in Section 10. Section 11 summarizes and gives directions for future work. Appendix A proves the soundness theorem.

2 The source language

The source language is a subset of Eiffel [8]. Its definition is the following:

```
exp                ::=  literal | var | exp op exp
stm                ::=  x := exp | stm; stm | from stm until exp loop stm end
                     |  if exp then stm else stm end
                     |  inspect x when $value_1$ then $s_1$ ... when $value_n$ then $s_n$ else $s_{n+1}$ end
                     |  check exp end
                     |  debug stm end
                     |  create {Type} x
                     |  x := y.Type@a
                     |  y.Type@a := exp
                     |  x := y.Type : routine_name( exp )
                     |  retry
once_routine       ::=  name (var : Type) : Type is
                          require exp
                          [ local var : Type, ... ]
                          once
                            stm
                          [ rescue
                            stm ]
                          ensure exp
                        end
non_once_routine   ::=  name (var : Type) : Type is
                          require exp
                          [ local var : Type, ... ]
                          do
                            stm
                          [ rescue
                            stm ]
                          ensure exp
                        end
routine            ::=  once_routine  |  non_once_routine
```

Expressions are side-effect-free and cannot throw an exception.

3 The Eiffel Program Logic

Our logic is based on the programming logics presented in [10, 16, 17]. We adapted them to Eiffel and we proposed new rules for Eiffel's instructions. The new rules are for *rescue* and *retry* instructions, multi-branch instruction, *check* and *debug* instructions, and once routines. In [17], a special variable χ is used to capture the status of the program such as normal or exceptional status. This variable is not necessary in the bytecode proof since non-linear control flow is implemented via jumps. To eliminate the χ variable, we use Hoare triples with two or three postconditions to encode the status of the program execution. This simplifies not only the translation but also the presentation.

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

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

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

The state of our Eiffel program 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 [15]. Following we present a short list of the operation we use for the object store.

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

In spite of Eiffel not having `continue` instructions, we need to treat jumps, because the `retry` instruction (inside of a `rescue` clause) allows one to jump to the beginning of the current routine. For example, we could use a `retry` instruction in the body of a loop and instead of jumping to the beginning of the loop as Java or C# (by using continue statements), we jump to the beginning of the routine.

The Eiffel logic is presented as follows: first the basic rules assign, conditional, multi-branch and compositional instructions are presented in subsection 3.1. The logic for loops is presented in subsection 3.2. Rescue and retry instruction are treated in subsection 3.3. The logic for check and debug instructions are presented in subsection 3.4 and 3.5. The logic for objects and method invocation is presented in subsection 3.6. The rules for routines are presented in subsection 3.7. We explain the semantic of once routines and present the corresponding logic in subsection 3.8. The language-independent rules are presented in subsection 3.9. Finally in subsection 3.10 we present two examples of the application of the logic.

3.1 Base Rules

Assign Instruction

$$\frac{}{\left\{\ P[e/x]\ \right\}\ \ x\ :=\ e\ \ \left\{\ P\ ,\ \mathit{false}\ ,\ \mathit{false}\ \right\}}$$

Conditional Instruction

$$\frac{\left\{\ P\ \wedge\ e\ \right\}\ s_1\ \left\{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \right\} \qquad \left\{\ P\ \wedge\ \neg e\ \right\}\ s_2\ \left\{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \right\}}{\left\{\ P\ \right\}\ \ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}\ \ \left\{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \right\}}$$

Multi-branch instruction

The multi-branch instruction supports a selection between a number of possible instructions. The order in which the branches are written does not influence the effect of the instruction. If the inspect expression (x) is equal to a value_i then the s_i instruction is executed. The inspect expression must be of type *INTEGER*, *CHARACTER*, *STRING* or $\mathit{TYPE}[T]$ for some type T. The values in different branches are different [7]. We assume the expression x cannot throw an exception and it does not have side effects.

$$\frac{\begin{array}{c}\left\{\ P\ \wedge\ x = \mathit{value}_1\ \right\}\ s_1\ \left\{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \right\}\\ \ldots \\ \left\{\ P\ \wedge\ x = \mathit{value}_n\ \right\}\ s_n\ \left\{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \right\}\\ \left\{\ P\ \wedge\ x \neq \mathit{value}_1\ \wedge\ \ldots \wedge\ x \neq \mathit{value}_n\ \right\}\ s_{n+1}\ \left\{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \right\}\end{array}}{\left\{\ P\ \right\}\ \begin{array}{l}\texttt{inspect}\ x\\ \texttt{when}\ \mathit{value}_1\ \texttt{then}\ s_1\\ \ldots\\ \texttt{when}\ \mathit{value}_n\ \texttt{then}\ s_n\\ \texttt{else}\ s_{n+1}\\ \texttt{end}\end{array}\ \left\{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \right\}}$$

We can extend the rule and assume that the value_i is a list of values. We only need to extend the premises $\{\ P\ \wedge\ x = \mathit{value}_1\ \}\ \ s_1\ \ \{\ Q\ \}$ to $\{\ P\ \wedge\ x = \mathit{value}_{1_1}\ \wedge\ x = \mathit{value}_{1_2} \ldots \wedge\ x = \mathit{value}_{1_m} \}\ \ s_1\ \ \{\ Q\ \}$ and the otherwise clause to $\{\ P\ \wedge\ (x \neq \mathit{value}_{1_1}\ \wedge\ \ldots\ x \neq \mathit{value}_{1_m}) \ldots\ \wedge\ (x \neq \mathit{value}_{n_1} \wedge x \neq \mathit{value}_{n_m}\ \}\ \ s_{n+1}\ \ \{\ Q\ \}$

Composition Instruction

In the compositional statement, the statement s_1 is executed first. The statement s_2 is executed if and only if s_1 has terminated normally.

$$\frac{\left\{\ P\ \right\}\ s_1\ \left\{\ Q_n\ ,\ R_r\ ,\ R_e\ \right\} \qquad \left\{\ Q_n\ \right\}\ s_2\ \left\{\ R_n\ ,\ R_r\ ,\ R_e\ \right\}}{\left\{\ P\ \right\}\ \ s_1 ; s_2\ \ \left\{\ R_n\ ,\ R_r\ ,\ R_e\ \right\}}$$

3.2 Loops

First, the instruction s_1 is executed. If s_1 either throws an exception or executes a `retry` instruction, then the postcondition of the loop is the postcondition of s_1 (Q_r for `retry` and R_e for exceptions). If s_1 terminates normally, then the body of the loop (s_2) is executed. If s_2 terminates normally then the invariant I_n holds. If s_2 executes a `retry` instruction then Q_r holds. If s_2 throws an exception, R_e holds.

$$\frac{\left\{\ P\ \right\}\ s_1\ \left\{\ I_n\ ,\ Q_r\ ,\ R_e\ \right\} \qquad \left\{\ \neg e\ \wedge\ I_n\ \right\}\ s_2\ \left\{\ I_n\ ,\ Q_r\ ,\ R_e\ \right\}}{\left\{\ P\ \right\}\ \texttt{from}\ s_1\ \texttt{invariant}\ I_n\ \texttt{until}\ e\ \texttt{loop}\ s_2\ \texttt{end}\ \left\{\ (I_n\ \wedge\ e)\ ,\ Q_r\ ,\ R_e\ \right\}}$$

3.3 Rescue and Retry Logic

retry rule

This rule sets the normal and exception postcondition to false and the `retry` postcondition to P due to the execution of the `retry` instruction.

$$\frac{}{\left\{\ P\ \right\}\ \texttt{retry}\ \left\{\ \mathit{false}\ ,\ P\ ,\ \mathit{false}\ \right\}}$$

rescue rule

This rule is defined for routines with `rescue` clause. If s_1 terminates normally, then the postcondition of the rule is the postcondition of s_1 (Q_n). The `rescue` block can produce two results: either s_2 is executed and terminates in a `retry` status and I_n holds or s_2 is executed and terminates normally or exceptional status and R_e holds. If the instruction s_2 executes an `retry` instruction, then control flow is transferred to the beginning of the routine and I_n holds. If s_2 terminates normally, the exception of s_1 is re-throw. If both (s_1 and s_2) throw an exception, the last one takes precedence.

$$\frac{\begin{array}{l} P\ \Rightarrow\ I_n \\ \left\{\ I_n\ \right\}\ s_1\ \left\{\ Q_n\ ,\ \mathit{false}\ ,\ I'_n\ \right\} \\ \left\{\ I'_n\ \right\}\ s_2\ \left\{\ R_e\ ,\ I_n\ ,\ R_e\ \right\} \end{array}}{\left\{\ P\ \right\}\ s_1\ \texttt{rescue}\ s_2\ \left\{\ Q_n\ ,\ \mathit{false}\ ,\ R_e\ \right\}}$$

This rule shows that a `rescue` and `retry` instruction is a loop. It iterates over $s_1; s_2$ until no exception is thrown in s_1 or s_2 does not executes a `retry` instruction. I_n is the invariant of the loop. The normal postcondition is Q_n because if s_1 is excuted and terminates normally, then the `rescue` clause is not executed.

3.4 Check instruction

The `check` instruction helps to express a property that you believe will be satisfied. If the property is satisfied then the system does not change. If the property is not satisfied then an exception is thrown.

$$\frac{}{\left\{\ P\ \right\}\ \texttt{check}\ e\ \texttt{end}\ \left\{\ (P\ \wedge\ e)\ ,\ \mathit{false}\ ,\ (P\ \wedge\ \neg e)\ \right\}}$$

3.5 Debug instruction

Let $\mathcal{D}$ be a program variable used to describe the status of `debug`. The possible values of $\mathcal{D}$ are: *not_debug* and *debug*. $\mathcal{D} = \textit{not\_debug}$ means that the user has defined the debug option of the class as off. $\mathcal{D} = \textit{debug}$ means that the user has defined the debug option of the class as on. The logic for debug instructions consist of two rules: one when the debug option is on and another when the debug option is off.

$$\frac{\{\ P\ \}\quad s_1\quad \{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \}}{\left\{\begin{array}{l}(P\ \wedge\ \mathcal{D} = \textit{debug})\ \vee\\ (P'\ \wedge\ \mathcal{D} = \textit{not\_debug})\end{array}\right\}\ \texttt{debug}\ s_1\ \texttt{end}\ \{\ Q_n\ \vee\ P'\ ,\ Q_r\ ,\ Q_e\ \}}$$

3.6 Creation Instruction, read and write attribute, and routine invocation

This section presents the adaptation of new, read field, write field and invocations rules from [10, 16] to Eiffel.

Creation Instruction

$$\frac{\{\ P\ \}\quad T@\textit{make}(p)\quad \{\ Q_n\ ,\ Q_e\ \}}{\left\{ P\left[\begin{array}{l}\textit{new}(\$, T)/\textit{Current},\\ \$<T>/\$,\\ e/p\end{array}\right]\right\}\ x := \texttt{create}\ \{T\}.\textit{make}(e)\ \{\ Q_n[x/\textit{Current}]\ ,\ \textit{false}\ ,\ Q_e[x/\textit{Current}]\ \}}$$

Read Attribute

$$\left\{\begin{array}{l}(y \neq \textit{Void}\ \wedge\ P[\$(\textit{instvar}(y, S@a))/x])\ \vee\\ (y = \textit{Void}\ \wedge\ Q_e\left[\begin{array}{l}\$<\textit{NullPExc}>/\$,\\ \textit{new}(\$, \textit{NullPExc})/\textit{excV}\end{array}\right])\\ \end{array}\right\}\ x := y.S@a\ \left\{\ P\ ,\ \textit{false}\ ,\ Q_e\ \right\}$$

Write Attribute

$$\left\{\begin{array}{l}(y \neq \textit{Void}\ \wedge\ P[\$<\textit{instvar}(y, S@a) := e>/\$])\ \vee\\ (y = \textit{Void}\ \wedge\ Q_e\left[\begin{array}{l}\$<\textit{NullPExc}>/\$,\\ \textit{new}(\$, \textit{NullPExc})/\textit{excV}\end{array}\right])\\ \end{array}\right\}\ y.S@a := e\ \left\{\ P\ ,\ \textit{false}\ ,\ Q_e\ \right\}$$

Invocation

$$\frac{\{\ P\ \}\quad T:m(p)\quad \{\ Q_n\ ,\ \textit{false}\ ,\ Q_e\ \}}{\left\{\begin{array}{l}(y \neq \textit{Void}\ \wedge\ P[y/\textit{Current}, e/p])\ \vee\\ (y = \textit{Void}\ \wedge\ Q_e\left[\begin{array}{l}\$<\textit{NullPExc}>/\$,\\ \textit{new}(\$, \textit{NullPExc})/\textit{excV}\end{array}\right])\\ \end{array}\right\}\ x := y.T:m(e)\ \{\ Q_n[x/\textit{result}]\ ,\ \textit{false}\ ,\ Q_e\ \}}$$

3.7 Routines Rules

Routines implementation

$$\frac{\{\ P\ \}\ \ body(T:m)\ \ \{\ Q_n\ ,\ false\ ,\ Q_e\ \}}{\{\ P\ \}\ \ T:m\ \ \{\ Q_n\ ,\ false\ ,\ Q_e\ \}}$$

Subtype rule

$$\frac{\begin{array}{l} S \preceq T \\ \left\{\ P\ \right\}\ \ S:m\ \ \left\{\ Q_n\ ,\ false\ ,\ Q_e\ \right\} \end{array}}{\left\{\ \tau(Current) \preceq S\ \wedge\ P\ \right\}\ \ T:m\ \ \left\{\ Q_n\ ,\ false\ ,\ Q_e\ \right\}}$$

Local rule

$$\frac{\left\{\ P\ \wedge\ v_1 = init(T_1)\ \wedge\ ... \wedge\ v_n = init(T_n)\ \right\}\ \ s\ \ \left\{\ Q_n\ ,\ false\ ,\ Q_e\ \right\}}{\left\{\ P\ \right\}\ \ \texttt{local}\ \ v_1 : T_1;\ ...\ v_n : T_n;\ s\ \ \left\{\ Q_n\ ,\ false\ ,\ Q_e\ \right\}}$$

where P does not refer to $v_1, ..., v_n$.

3.8 Once routines: once procedures and once functions

This section presents the logic for once routines. First, we present the logic for once procedure and afterwards we extend it for once functions.

3.8.1 Once procedures

A once procedure is a procedure that is executed only once. The first invocation executes the body of the procedure. However, the remaining executions do nothing. Let p be an once procedure defined in the class C as:

```
  p (i: T) is
              an once procedure
     once
         body
     end
```

To define the logic for once procedures, we introduce two fresh variables: C_p_done (of boolean type) and $C\_p\_exception$ (of type **EXCEPTION**). $C\_p\_done = true$ expresses that the procedure p in the class C has been executed at least one time. $C\_p\_done = false$ represents that the procedure has never been executed. $C\_p\_exception$ is used to express whether an exception has been thrown in the procedure p or not/ It also stores the type of the exception.

For readers not familiar with once routines in Eiffel, following we present a translation to Java [1]. This translation also illustrates all the cases we have to consider in the definition of the logic for once procedures.

[1]The variables C_p_done and C_p_exception are declared in a class DATA and they are static variables.

```
if ( ! DATA.C_p_done) {
    DATA.C_p_done = true;
    try {
        Eiffel2Java(body)
    }
    catch (Exception e) {
        DATA.C_p_exception = e;
        throw e;
    }
}
if (DATA.C_p_exception != null) {
    throw DATA.C_p_exception;
}
```

The execution of a once routine can produce different results depending on whether it is the first execution of the once routine or not and whether an exception was thrown or not. We present them in four cases:

- If it is the first execution of the once routine, then the body of the routine is executed and produces its result without throwing an exception.
- If it is the first execution of the once routine, then the body of the routine is executed and produces an exception. The exception is stored to be able to reproduce the same exception during the next executions of the routine.
- If it is not the first execution of the once routine and the first execution produced an exception then the same exception is returned.
- If it is not the first execution of the once routine and the first execution did not produce an exception then it does nothing.

Let S be the following precondition

$$S \equiv \left\{ \begin{array}{l} (\neg C\_p\_done \;\wedge\; P) \vee \\ \Big(\; C\_p\_done \;\wedge\; P' \;\wedge\; C\_p\_exception = Void \; \Big) \vee \\ (C\_p\_done \;\wedge\; P'' \;\wedge\; C\_p\_exception \neq Void) \end{array} \right\}$$

The rule is defined as follows:

$$\frac{\left\{ \begin{array}{l} P[false/C\_p\_done] \wedge \\ C\_p\_done \end{array} \right\} \; body(C:p) \; \left\{ (Q_n \;\wedge\; C\_p\_done) \;,\; false \;,\; (Q_e \;\wedge\; C\_p\_done) \right\}}{\left\{ \; S \; \right\} \;\; C:p(i) \; \left\{ \left(\begin{array}{l} C\_p\_done \;\wedge \\ C\_p\_exception = Void \;\wedge \\ (Q_n \vee P') \end{array} \right) , \; false \; , \left(\begin{array}{l} C\_f\_done \;\wedge \\ C\_f\_exception \neq Void \;\wedge \\ (Q_e \;\vee P'') \end{array} \right) \right\}}$$

3.8.2 Once functions

The difference between once procedures and once functions is that once functions return a value and procedures do not. The returned value of once functions is always the same value (which was obtained in the first invocation).

Let f be an once function defined in the class C as:

```
f (i: T1): T2 is
        an once function
    once
        body
    end
```

Besides the variables C_f_done and $C\_f\_exception$ introduced in the above Section, we use a fresh variable C_f_result. Its type is the return type of the once function (T_2 for the function f). C_f_result is used to store the returned value of the once function.

Similar to once procedures, we present the translation of once functions to Java. We assume the result of the body of the function is assigned to C_f_result in *Eiffel2Java(body)*.

```
if ( ! DATA.C_f_done) {
    DATA.C_f_done = true;
    try {
        Eiffel2Java(body)
    }
    catch (Exception e) {
        DATA.C_f_exception = e;
        throw e;
    }
}
if (DATA.C_f_exception != null) {
    throw DATA.C_f_exception;
}
else {
    return DATA.C_f_result;
}
```

Let S be the following a precondition

$$S \equiv \left\{ \begin{array}{l} (\neg C\_f\_done \ \wedge \ P) \vee \\ \left(\begin{array}{l} C\_f\_done \ \wedge \ P' \ \wedge \ C\_f\_result = C\_F\_RESULT \ \wedge \\ C\_f\_exception = Void \end{array} \right) \vee \\ (C\_f\_done \ \wedge \ P'' \ \wedge \ C\_f\_exception \neq Void) \end{array} \right\}$$

The rule is defined as follows:

$$\frac{\left\{ \begin{array}{l} P[false/C\_f\_done] \wedge \\ C\_f\_done \end{array} \right\} \ body(C:f) \ \left\{ (Q_n \ \wedge \ C\_f\_done) \ , \ false \ , \ (Q_e \ \wedge \ C\_f\_done) \right\}}{\left\{ \ S \ \right\} \ C:f(i) \ \left\{ \begin{array}{l} \left(\begin{array}{l} C\_f\_done \ \wedge \ C\_f\_exception = Void \ \wedge \\ (Q_n \vee \left(\begin{array}{l} P' \ \wedge \ result = C\_F\_RESULT \ \wedge \\ C\_f\_result = C\_F\_RESULT \end{array} \right) \end{array} \right) , \\ \\ false \ , \\ \\ \left(\begin{array}{l} C\_f\_done \ \wedge \ C\_f\_exception \neq Void \ \wedge \\ (Q_e \ \vee P'') \end{array} \right) \end{array} \right\}}$$

3.8.3 Discussion

The above rule allows us to write a proof whatever once function we write (even recursive once functions). The rule is correct and models all possible cases of once functions.

However, verify a program which contains a once function is not simple. We have to know the execution path of every once function so that we know whether use the result of the first invocation or invoke it by first time. Furthermore, the verification technique is not modular. We need to add the information of *C_f_done* and *C_f_result* for all routine that invokes a once function. This makes the logic hard to use.

To solve this problem, we have analyzed Eiffel libraries looking for good once routines. We have found that there hardly exist once procedures and that most of once functions do not have parameters. Furthermore, most of once functions are used to either create a shared object and return it or to execute an expensive calculation.

We have proposed a new rule for once functions. It assumes once functions are:

- state independent,
- they do not have arguments, and
- they are pure functions (side effect free)

With these assumptions, we do not need to know whether the once function was executed once or not. The function returns always the same result.

The rule is defined as follows:

$$\frac{\left\{\ P\ \right\}\ \ \mathit{body}(C:f)\ \ \left\{\ Q_n\ ,\ \mathit{false}\ ,\ Q_e\ \right\}}{\left\{\ P\ \right\}\ \ C:f(i)\ \ \left\{\ Q_n\ \wedge\ \mathit{result} = \mathit{cache}^f\ ,\ \mathit{false}\ ,\ Q_e\ \right\}}$$

where cache^f stores the result of the once function f.

3.9 Language-independent rules

False axiom

$$\frac{}{\left\{\ \mathit{false}\ \right\}\ s_1\ \left\{\ \mathit{false}\ ,\ \mathit{false}\ ,\ \mathit{false}\ \right\}}$$

Strength

$$\frac{\begin{array}{l} P' \Rightarrow P \\ \left\{\ P\ \right\}\ s_1\ \left\{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \right\} \end{array}}{\left\{\ P'\ \right\}\ s_1\ \left\{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \right\}}$$

Weak

$$\frac{\begin{array}{l} \left\{\ P\ \right\}\ s_1\ \left\{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \right\} \\ Q_n \Rightarrow Q'_n \\ Q_r \Rightarrow Q'_r \\ Q_e \Rightarrow Q'_e \end{array}}{\left\{\ P\ \right\}\ s_1\ \left\{\ Q'_n\ ,\ Q'_r\ ,\ Q'_e\ \right\}}$$

Invariant

$$\frac{\left\{\ P\ \right\}\ s_1\ \left\{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \right\}}{\left\{\ P \wedge W\ \right\}\ s_1\ \left\{\ Q_n \wedge W\ ,\ Q_r \wedge W\ ,\ Q_e \wedge W\ \right\}}$$

Substitution

$$\frac{\left\{\ P\ \right\}\ s_1\ \left\{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \right\}}{\left\{\ P[t/Z]\ \right\}\ s_1\ \left\{\ Q_n[t/Z]\ ,\ Q_r[t/Z]\ ,\ Q_e[t/Z]\ \right\}}$$

Conjunction

$$\frac{\begin{array}{c} \left\{\ P^1\ \right\}\ s_1\ \left\{\ Q^1_n\ ,\ Q^1_r\ ,\ Q^1_e\ \right\} \\ \\ \left\{\ P^2\ \right\}\ s_1\ \left\{\ Q^2_n\ ,\ Q^2_r\ ,\ Q^2_e\ \right\} \end{array}}{\left\{\ P^1 \wedge P^2\ \right\}\ s_1\ \left\{\ Q^1_n \wedge Q^2_n\ ,\ Q^1_r \wedge Q^2_r\ ,\ Q^1_e \wedge Q^2_e\ \right\}}$$

Disjunction

$$\frac{\begin{array}{c} \left\{\ P^1\ \right\}\ s_1\ \left\{\ Q^1_n\ ,\ Q^1_r\ ,\ Q^1_e\ \right\} \\ \\ \left\{\ P^2\ \right\}\ s_1\ \left\{\ Q^2_n\ ,\ Q^2_r\ ,\ Q^2_e\ \right\} \end{array}}{\left\{\ P^1 \vee P^2\ \right\}\ s_1\ \left\{\ Q^1_n \vee Q^2_n\ ,\ Q^1_r \vee Q^2_r\ ,\ Q^1_e \vee Q^2_e\ \right\}}$$

all-rule

$$\frac{\left\{\ P[Y/Z]\ \right\}\ s_1\ \left\{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \right\}}{\left\{\ P[Y/Z]\ \right\}\ s_1\ \left\{\ \forall Z : Q_n\ ,\ \forall Z : Q_r\ ,\ \forall Z : Q_e\ \right\}}$$

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

ex-rule

$$\frac{\left\{\ P[Y/Z]\ \right\}\ s_1\ \left\{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \right\}}{\left\{\ P[Y/Z]\ \right\}\ s_1\ \left\{\ \exists Z : Q_n\ ,\ \exists Z : Q_r\ ,\ \exists Z : Q_e\ \right\}}$$

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

3.10 Application

In this section we present two examples of the application of the Eiffel logic. In subsection 3.10.1 we present an example of rescue and retry rules. The application of once routines is presented in subsection 3.10.2.

3.10.1 Application of rescue and retry rules

This example is a very simple calculator with multiplication and division operations. It requires that the second operator is different from zero to avoid exceptions. But before applying the operation, the calculator tries to open a file. Opening a file can throw an exception (for example, because the file does not exist). For operation 1 (division) it tries to open the file three times, if it cannot open the file after the third attempt, it does not try again and it applies the division. In the case of multiplication (operation/=1) it tries to open the file and if the file cannot be opened, it throws an exception and the exception is propagated. To facilitate the reading, we add the text *normal* and *exc* to indicate that the postcondition is a normal and exceptional postcondition respectively.

Following, we present the source proof for the example of figure 1.

Table 1: Proof for the example of figure 1.

```
calculator (op1, op2, operation: INTEGER):INTEGER is
    require
        not_zero: op2 /= 0
    local
        attempt: INTEGER
    do
        if (operation=1) then
            if (attempt < 3) then
                open_a_file
            end
            Result:=op1 // op2
        else
            open_a_file
            Result:=op1 * op2
        end
    rescue
        if (operation=1) then
            attempt:=attempt+1
            retry
        end
        Result:=1
    end
```

Figure 1: A simple example of rescue and retry instructions.

```
calculator (op1, op2, operation: INTEGER):INTEGER is
    require
        not_zero: op2 /= 0
    local
        attempt: INTEGER
    do
        $\{ \ op_2 \neq 0 \ \wedge \ attempt = 0 \ \}$
        $\Rightarrow$
        $\{ \ op_2 \neq 0 \ \wedge \ (operation = 1 \ \Rightarrow \ (attempt \geq 0 \ \wedge \ attempt \leq 3)) \ \}$
        if (operation=1) then
            $\{ \ op_2 \neq 0 \ \wedge \ attempt \geq 0 \ \wedge \ attempt \leq 3 \ \wedge \ operation = 1 \ \}$
            if (attempt < 3) then
                $\{ \ op_2 \neq 0 \ \wedge \ attempt \geq 0 \ \wedge \ attempt < 3 \ \wedge \ operation = 1 \ \}$
                open_a_file
                $\left\{ \begin{array}{l} normal: \ op_2 \neq 0 \ \wedge \ attempt \geq 0 \ \wedge \ attempt < 3 \ \wedge \ operation = 1, \\ exc: \ op_2 \neq 0 \ \wedge \ attempt \geq 0 \ \wedge \ attempt < 3 \ \wedge \ operation = 1 \end{array} \right\}$
            end
            $\left\{ \begin{array}{l} normal: \ op_2 \neq 0 \ \wedge \ attempt \geq 0 \ \wedge \ attempt < 3 \ \wedge \ operation = 1 \ , \\ exc: \ op_2 \neq 0 \ \wedge \ attempt \geq 0 \ \wedge \ attempt < 3 \ \wedge \ operation = 1 \end{array} \right\}$
            Result:=op1 // op2
            $\left\{ \begin{array}{l} normal: \left( \begin{array}{l} op_2 \neq 0 \ \wedge \ attempt \geq 0 \ \wedge \ attempt < 3 \\ \wedge \ Result = op1//op2 \ \wedge \ operation = 1 \end{array} \right) , \\ exc: (op_2 \neq 0 \ \wedge \ attempt \geq 0 \ \wedge \ attempt < 3 \ \wedge \ operation = 1) \end{array} \right\}$
        else
```

Continued on next page

```
                $\{\ op_2 \neq 0\ \wedge\ operation \neq 1\ \}$
                open_a_file
                $\left\{\begin{array}{l} normal:\ op_2 \neq 0\ \wedge\ operation \neq 1\ , \\ exc:\ op_2 \neq 0\ \wedge\ operation \neq 1 \end{array}\right\}$
                Result:=op1 * op2
                $\left\{\begin{array}{l} normal:\ op_2 \neq 0\ \wedge\ operation \neq 1\ \wedge\ Result = op1 * op2\ , \\ exc:\ op_2 \neq 0\ \wedge\ operation \neq 1 \end{array}\right\}$
            end
        rescue
            $\left\{ normal: \left( \begin{array}{l} (operation = 1 \wedge\ op_2 \neq 0\ \wedge\ attempt \geq 0\ \wedge\ attempt < 3) \vee \\ (operation \neq 1 \wedge\ op_2 \neq 0) \end{array} \right) \right\}$
            if (operation=1) then
                $\{\ normal:\ operation = 1 \wedge\ op_2 \neq 0\ \wedge\ attempt \geq 0\ \wedge\ attempt < 3\ \}$
                attempt:=attempt+1
                $\{\ normal:\ operation = 1 \wedge\ op_2 \neq 0\ \wedge\ attempt \geq 0\ \wedge\ attempt \leq 3\ \}$
                retry
                $\{\ retry:\ operation = 1 \wedge\ op_2 \neq 0\ \wedge\ attempt \geq 0\ \wedge\ attempt \leq 3\ \}$
            end
            $\left\{\begin{array}{l} normal:\ operation \neq 1 \wedge\ op_2 \neq 0 \\ retry:\ operation = 1 \wedge\ op_2 \neq 0\ \wedge\ attempt \geq 0\ \wedge\ attempt \leq 3 \end{array}\right\}$
            Result:=-1
            $\left\{\begin{array}{l} normal:\ operation \neq 1 \wedge\ op_2 \neq 0\ \wedge\ Result = -1 \\ retry:\ operation = 1 \wedge\ op_2 \neq 0\ \wedge\ attempt \geq 0\ \wedge\ attempt \leq 3 \end{array}\right\}$
    end
            $\left\{\begin{array}{l} normal: \left( \begin{array}{l} (operation = 1 \wedge\ Result = op1//op2) \vee \\ (operation \neq 1 \wedge\ Result = op1 * op2) \end{array} \right) \\ exc:\ (operation \neq 1 \wedge\ op_2 \neq 0\ \wedge\ Result = -1) \end{array}\right\}$
```

3.10.2 Application of Once functions

In this section we present a simple example of the application of the logic for once functions. It invokes a once function two times with different parameters and it shows that the result is always the same. Figure 2 presents the Eiffel program and table 2 presents its proof.

Table 2: Proof for the example of figure 2.

```
    f (p: INTEGER): INTEGER is
            - - an example of once function
        once
            $\{\ true\ \}$
            Result := p * 2
            $\{\ Result = p * 2\ \}$
        ensure
            Result = p * 2
        end
```

Continued on next page

```
  f (p: INTEGER): INTEGER is
              an example of once function
      once
          Result := p * 2
      ensure
          Result = p * 2
      end

  my_example is
              invoke the function f
      do
          x := f(5)
          x2 := f(8)
      end
```

Figure 2: Source once function.

```
my_example is
        - - invoke the function f
    do
        { ¬C_f_done }
        x := f(5)
        { x = 5 * 2 ∧ C_f_done ∧ C_f_exception = Void ∧ C_f_result = x }
        x2 := f(8)
        { x = 10 ∧ C_f_done ∧ C_f_exception = Void ∧ C_f_result = 10 ∧ x2 = C_f_result }
        ⇒
        { x = 10 ∧ C_f_result = 10 ∧ C_f_exception = Void ∧ x2 = 10 }
    end
```

4 CIL Kernel Language

The CIL language consists of classes with fields and methods. The methods are implemented as method bodies consisting of a sequence of labelled intermediate instructions. The instructions are executed indirectly by means of a Just-in-Time Compiler (JIT). The JIT translates the instructions into machine code for the particular computer on which the program is to be executed.

The instructions are executed in an *abstract stack machine*. The instruction set consists of instructions that *push* operands on the abstract evaluation stack, instructions that operate on the top of stack operands, and instructions that *pop* operands off the stack and store them in memory or in local variables.

There are about 220 instructions in CIL. Following, we present an informal description of the instructions we use to translate Eiffel to CIL (more detail about CIL see [5]).

- ldc v: pushes a number constant v onto the stack.
- ldstr v: pushes a string constant v onto the stack.
- ldnull: pushes null onto the stack.
- ldloc x: pushes the value of a local variable x onto the stack.
- ldarg x: pushes the value of a method parameter x onto the stack.
- stloc x: pops the top element off the stack and assigns it to the local variable x.

- starg x: pops the top element off the stack and assigns it to the method argument 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. We consider the following binary instruction:
  1. add, rem, mul, div: take 2 input values, it removes the 2 top elements from the stack applying the addition, subtraction, multiplication or division operation resp. and put the result onto the stack.
  2. ceq, clt, cgl: take 2 input values, it removes the 2 top elements from the stack applying the equal, less than, great than operation resp. and put the result onto the stack.
- br l: transfers the control program to the point l.
- brtrue l: transfers the control program to the point l if the top element of the stack is true and unconditionally pops it.
- brfalse l: transfers the control program 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 instance void Class::.ctor(). T: allocates a new object of type T and pushes it onto the stack.
- callvirt 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, callvirt invokes virtual methods. The code depends on the actual type of the object reference (dynamic dispatch).
- ldfld F: replaces the top element by its field F (an instance field).
- ldsfld F: replaces the top element by its field F (a static field).
- stfld F: sets the field F (an instance field) of the object denoted by the second-topmost element to the top element of the stack and pops both values.
- stsfld F: sets the field F (a static field) of the object denoted by the second-topmost element to the top element of the stack and pops both values.
- ret return to caller
- nop: has no effect.

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^1_p is shown in figure 3. 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 → 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.:

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

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

6 Proof transformation from Eiffel to CIL

Our proof-transforming compiler is based on transformation functions, ∇_S and ∇_E, for instructions and expressions respectively. Both functions yield a sequence of Bytecode instructions and their specification. ∇_S generates this sequence from a proof for a source instruction and ∇_E 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:

$$\begin{array}{ll} \nabla_E & : \mathit{Precondition} \times \mathit{Expression} \times \mathit{Postcondition} \times \mathit{Label} \Rightarrow \mathit{Bytecode\_Proof} \\ \nabla_S & : \mathit{Proof\_Tree} \times \mathit{Label} \times \mathit{Label} \times \mathit{Label} \times \mathit{Label} \Rightarrow \mathit{Bytecode\_Proof} \end{array}$$

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

I_l	$wp^1_p(I_l)$
ldc v	$unshift(E_{l+1}[v/s(0)])$
ldstr v	$unshift(E_{l+1}[v/s(0)])$
ldnull	$unshift(E_{l+1}[null/s(0)])$
ldloc x	$unshift(E_{l+1}[x/s(0)])$
ldarg x	$unshift(E_{l+1}[x/s(0)])$
stloc x	$(shift(E_{l+1}))[s(0)/x]$
starg x	$(shift(E_{l+1}))[s(0)/x]$
bin_{op}	$(shift(E_{l+1}))[s(1)\,op\,s(0)/s(1)]$
br 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	$unshift(E_{l+1}[new(\$, T)/s(0), \$ < T > /\$]$
ldfld $T@a$	$E_{l+1}[\$(iv(s(0), T@a))/s(0)] \ \wedge \ s(0) \neq null$
stfld $T@a$	$(shift^2(E_{l+1}))[\$ < iv(s(1), T@a) := s(0) > /\$] \ \wedge \ s(1) \neq null$
ret	true
aret	(shift(Q))[s(0)/result] where Q is the method's postcondition.
nop	E_{l+1}

Figure 3: The values of the wp^1_p function.

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

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.

In ∇_S the four labels are: *start*, *next*, *retry* and *exc*. *start* and *next* are used to know the starting label and the next label of the translation. For example in the translation of `if then else` instructions, the next label is used to know where to jump after the end of the else translation (after the then part). We could eliminate the *next* label introducing `nop` instructions after every *if*. The *retry* is used to process a `retry` instruction. It means that control flow will be transferred to the label *retry* when a `retry` instruction is processed (it represents the beginning of the routine being processed). *exc* is used to store the label where to jump if an exception is throw. This label is used in the soundness theorem.

The *BytecodeProof* type is defined as a list of *InstrSpec*, where *InstrSpec* is an instruction specification. The translation functions are based on the definition of wp^1_p (figure 3). Each translation was derived using the object-oriented source proof and the wp^1_p definition.

In the following, we present the proof translation. Table 3 comprises the naming conventions we use in the rest of this paper. Section 6.3 presents the translation for expressions. Section 6.4 describes the translation for language-independent rules. Section 6.5 describes the translation for assignment, conditional and compositional instructions. The translation for create features and

Type	Typical use
$Precondition \cup Postcondition$	P, Q, R, U, V
$Normal\ Postcondition$	Q_n, R_n
$Retry\ Postcondition$	Q_r, R_r
$Exception\ Postcondition$	Q_e, R_e
$Label$	$l_{start}, l_{next}, l_{retry}, l_{exc}$
	$l_b, l_c, ..., l_g$

Table 3: Naming conventions.

read and write attribute is illustrated in Section 6.6. The translation for routines is presented in Section 6.7. Finally, Section 6.8 presents the translation for instructions specific of Eiffel.

6.1 Compiling Eiffel to CIL

The Eiffel's object model differs from the CIL's object model. Eiffel supports concepts like *multiple inheritance* which CIL does not. These concepts can be modeled in CIL but in a different way than in Eiffel.

When an Eiffel class is compiled to CIL, four classes are generated. For example, the Eiffel class *MY_CLASS* is compiled to an interface *MyClass*, and tree classes: *Impl.MyClass*, *Create.MyClass*, and *Data.MyClass* in CIL. The class *Impl.MyClass* implements the interface *MyClass*. The class *Create.MyClass* is used to compile creation procedures. And the class *Data.MyClass* is used to compile `once` routines.

Creation procedures

Eiffel allows to write creation procedures with the same parameters by renaming or given another name. However, languages like Java or C# do not. CIL defines constructors using the class name and does not allow to define several constructors with the same parameters.

The compilation of the creation procedure $make(v_1 : T_1, ...v_n : T_n)$ defined in the Eiffel class *MY_CLASS* produces four methods in CIL:

- An abstract method $make(v_1 : T_1, ...v_n : T_n)$ in the interface *MyClass*,
- A method $make(v_1 : T_1, ...v_n : T_n)$ in the class *Impl.MyClass*,
- A method $make(v_0 : MyClass, v_1 : T_1, ...v_n : T_n)$ in the class *Impl.MyClass*, and
- A method $make(v_1 : T_1, ...v_n : T_n)$ in the class *Create.MyClass*.

The method $make(v_1 : T_1, ...v_n : T_n)$ in the class *Impl.MyClass* is defined as follows:

```
.method public void make(v_0:MyClass, v_1:T_1,...v_n:T_n) cil managed
  {
    IL_0000:  ldarg.0
    IL_0001:  ldarg.1     // load the argument v_1
              ...
    IL_000n:  ldarg.n     // load the argument v_n
    IL_0002:  call        void Impl.MyClass::make(MyClass, T_1, ... T_n)
    IL_0007:  ret
  }
```

The body of the method $make(v_0 : MyClass, v_1 : T_1, ...v_n : T_n)$ contains the creation procedures' implementation. The method $make(v_1 : T_1, ...v_n : T_n)$ in the class *Create.MyClass* is defined as follows:

```
.method public void make(v_1:T_1,...v_n:T_n) cil managed
    {
      IL_0000:  newobj    void Impl.MyClass::.ctor()
      IL_0005:  dup
      IL_0006:  ldarg.0  // load the argument v_1
                 ...
      IL_000n:  ldarg.n  // load the argument v_n
      IL_0007:  callvirt    void Impl.MyClass::make(T_1, ... T_n)
      IL_000c:  ret
    }
```

Attributes

The *Uniform Access Principle* states that all services offered by a module should be available through an uniform notation, which does not depend whether they are implemented using an attribute or using a query that computes the result.

To compile an attribute *item: T* defined in the class *MyClass*, the followings methods and fields are created:

- *Interface MyClass*: a *.method abstract T item()* and a *.method abstract void set_item(T)* are defined.
- *Class Impl.MyClass*: a *.field public T item* is declared. The methods *item()* and *set_item(T)* are implemented as follows:

```
.method public T item() cil managed
    {
        IL_0000:  ldarg.0
        IL_0001:  ldfld        T Impl.MyClass::$$item
        IL_0006:  ret
    }
```

```
.method public void set_item(T i) cil managed
    {
        IL_0000:  ldarg.0
        IL_0001:  ldarg.1
        IL_0002:  stfld        T Impl.MyClass::$$item
        IL_0007:  ret
    }
```

Once Routines

To compile once routines, static fields are used. In the class *Data.MyClass* three static fields are defined:

- *.field public static bool name_done*
- *.field public static object name_exception*
- *.field public static T name_result*

where *name* is the name of the once routine and *T* its returned type. If the routine is a procedure, the last field is not declared. *name_done* is used to know whether the once routine was executed before or not; *name_exception* is used to know whether the once routine throws an exception or not and it also stores it object; and *name_result* is used to store the result.

Multiple Inheritance

Due to CIL does not support multiple inheritance, the compilation of a class *MyClass* generates an interface *MyClass* and a class *Impl.MyClass* which implements the interface. If in Eiffel, the class *MY_CLASS* inherits the classes C_1, C_2,...,C_n, then in CIL, the class *Impl.MyClass* extends the interfaces C_1, C_2,..., and C_n.

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 routines and its attributes. Each routine consists of a proof tree. The PTC takes the first class and for every routine of the class it creates a method in the interface and a method in the implementation class which body is the result of the translation function ∇_S setting the starting label to l_a (a symbolic label), the next label to l_b and the retry and exception label to $\emptyset$.

Per every creation procedure, the *make* methods are created as explained in the above section. The CIL proof are generated automatically. The CIL proof of these methods are the followings:

$.method\ public\ void\ make(v_0 : MyClass,\ v_1 : T_1, ...v_n : T_n)\ cil\ managed$

{

$\{P\}$ IL_{00}: ldarg.0

$\{P \ \wedge\ s(0) = this\}$ IL_{01}: ldarg.1

...

$\left\{ \begin{array}{l} P \ \wedge\ s(n-1) = this\ \wedge \\ s(n-2) = v_1\ \wedge ...s(0) = v_{n-1} \end{array} \right\}$ IL_{0n}: ldarg.n

$\left\{ \begin{array}{l} P \ \wedge\ s(n-1) = this\ \wedge \\ s(n-2) = v_1\ \wedge ...s(0) = v_n \end{array} \right\}$ IL_{07}: callvirt void Impl.MyClass::make(T_1, ...T_n)

$\{Q\}$ IL_{0c}: ret

}

$.method\ public\ void\ make(v_1 : T_1, ...v_n : T_n)\ cil\ managed$

{

$\{P\}$ IL_{0a} newobj void Impl.MyClass::.ctor()

$\{P \ \wedge\ s(0) = new(\$, MyClass)\}$ IL_{0b} dup

$\left\{ \begin{array}{l} P \ \wedge\ s(1) = new(\$, MyClass)\ \wedge \\ s(0) = new(\$, MyClass) \end{array} \right\}$ IL_{00}: ldarg.0

$\left\{ \begin{array}{l} P \ \wedge\ s(2) = new(\$, MyClass)\ \wedge \\ s(1) = new(\$, MyClass) \\ s(0) = this \end{array} \right\}$ IL_{01}: ldarg.1

...

$\left\{ \begin{array}{l} P \ \wedge\ s(n+1) = new(\$, MyClass)\ \wedge \\ s(n) = new(\$, MyClass) \\ \wedge\ s(n-1) = this\ \wedge \\ s(n-2) = v_1\ \wedge ...s(0) = v_{n-1} \end{array} \right\}$ IL_{0n}: ldarg.n

$\left\{ \begin{array}{l} P \ \wedge\ s(n+1) = new(\$, MyClass)\ \wedge \\ s(n) = new(\$, MyClass) \\ \wedge\ s(n-1) = this\ \wedge \\ s(n-2) = v_1\ \wedge ...s(0) = v_n \end{array} \right\}$ IL_{07}: callvirt void Impl.MyClass::make(T_1, ...T_n)

$\{Q\}$ IL_{0c}: ret

}

Per every attribute, the methods set and get are defined as we described in the above Section. The CIL proof is also generated. The CIL proof for the attribute *item* (presented above) is the following:

```
.method public T item() cil managed
 {
    {true}                              IL00: ldarg.0
    {s(0) = this}                       IL01: ldfld T Impl.MyClass::$$item
    {s(0) = $(iv(this, MyClass@item))}  IL06: ret
 }
.method public void set_item(T i) cil managed
 {
    {i ≠ null}                                IL00: ldarg.0
    {i ≠ null ∧ s(0) = this}                  IL00: ldarg.1
    {i ≠ null ∧ s(1) = this ∧ s(0) = i}       IL01: stfld T Impl.MyClass::$$item
    {$(this.@item) = i}                       IL06: ret
 }
```

In the following we present the translation rules. In Subsection 6.3 we present the expression translation. In Subsection 6.4, we describe the translation of language-independent rules for the source logic. In Subsection 6.5 we show the translation for *assign*, *conditional* and *composition* instructions. The translation of creation instructions, read and write attributes, and routine invocation are described in Subsection 6.6. Routines rules are translated in Subsection 6.7. Finally, we present the translation of instructions specific to Eiffel in Subsection 6.8.

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. To simplify the translation, we use stloc or ldloc instead of starg and ldarg when the variable is an argument.

Constants and variables are translated using ldloc. In the expressions' translation, first the expressions are pushed on the stack and then the instruction for the operation is added. The translations are defined as follows:

Constants

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

$$\{\mathrm{Q} \wedge \mathrm{unshift}(\mathrm{P}[c/s(0)])\} \quad l_a : \mathsf{ldloc}\ \mathrm{c}$$

Variables

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

$$\{\mathrm{Q} \wedge \mathrm{unshift}(\mathrm{P}[x/s(0)])\} \quad l_a : \mathsf{ldloc}\ \mathrm{x}$$

Expressions: e_1 **op** e_2

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

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

Expressions: unop e_2

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

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

6.4 Translation of language-independent Rules

In this section we present the translation of language-independent rules to CIL.

6.4.1 Strength

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

$$\nabla_S\left(\frac{\dfrac{\mathit{Tree}_1}{\{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_b\ ,\ Q_e\ \}}\qquad P' \Rightarrow P}{\{\ P'\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_b\ ,\ Q_e\ \}}\ ,\ l_{start}, l_{next}, l_{retry}, l_{exc}\right) =$$

$$\begin{array}{l}\{P'\}\quad l_a : \mathsf{nop}\\ \qquad\nabla_S\left(\dfrac{\mathit{Tree}_1}{\left\{\ P\ \right\}\ \ s_1\ \ \left\{\ Q_n\ ,\ Q_b\ ,\ Q_e\ \right\}}, l_b, l_{next}, l_{retry}, l_{exc}\right)\end{array}$$

6.4.2 Weak

Similar to strength rule, in weak rule we translate $Q_n \Rightarrow Q'_n$ by using nop

$$\nabla_S\left(\frac{\dfrac{\mathit{Tree}_1}{\{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_b\ ,\ Q_e\ \}}\qquad \begin{array}{c}Q_n \Rightarrow Q'_n\\ Q_b \Rightarrow Q'_b\\ Q_e \Rightarrow Q'_e\end{array}}{\{\ P\ \}\ \ s_1\ \ \{\ Q'_n\ ,\ Q'_b\ ,\ Q'_e\ \}}, l_{start}, l_{next}, l_{retry}, l_{exc}\right) =$$

$$\begin{array}{l}\qquad\nabla_S\left(\dfrac{\mathit{Tree}_1}{\left\{\ P\ \right\}\ \ s_1\ \ \left\{\ Q_n\ ,\ Q_b\ ,\ Q_e\ \right\}},\ \ l_{start}, l_b, l_{retry}, l_{exc}\right)\\ \{Q_n\}\quad l_b : \mathsf{nop}\end{array}$$

6.4.3 Invariant

$$\nabla_S\left(\frac{\dfrac{\mathit{Tree}_1}{\{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_b\ ,\ Q_e\ \}}}{\{\ P\ \wedge\ W\ \}\ \ s_1\ \ \{\ Q_n\ \wedge\ W\ ,\ Q_b\ \wedge\ W\ ,\ Q_e\ \wedge\ W\ \}},\ l_{start}, l_{next}, l_{retry}, l_{exc}\right) =$$

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

$$\nabla_S\left(\frac{\mathit{Tree}_1}{\{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_b\ ,\ Q_e\ \}},\ l_{start}, l_{next}, l_{retry}, l_{exc}\right)$$

6.4.4 Substitution

$$\nabla_S\left(\frac{\dfrac{\mathit{Tree}_1}{\{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_b\ ,\ Q_e\ \}}}{\{\ P[t/Z]\ \}\ \ s_1\ \ \{\ Q_n[t/Z]\ ,\ Q_b[t/Z]\ ,\ Q_e[t/Z]\ \}},\ l_{start}, l_{next}, l_{retry}, l_{exc}\right) =$$

As before, first we generate

$$\nabla_S\left(\frac{\mathit{Tree}_1}{\{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_b\ ,\ Q_e\ \}},\ l_{start}, l_{next}, l_{retry}, l_{exc}\right)$$

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

$$\frac{\mathit{Tree}_1}{\{\ P^1\ \}\ \ s_1\ \ \{\ Q_n^1\ ,\ Q_b^1\ ,\ Q_e^1\ \}}$$

and let T_b be

$$\frac{\mathit{Tree}_2}{\{\ P^2\ \}\ \ s_1\ \ \{\ Q_n^2\ ,\ Q_b^2\ ,\ Q_e^2\ \}}$$

$$\nabla_S\left(\frac{T_a \qquad\qquad T_b}{\left\{\ P^1\ \wedge\ P^2\ \right\}\ \ s_1\ \ \left\{\ Q_n^1\ \wedge\ Q_n^2\ ,\ Q_b^1\ \wedge\ Q_b^2\ ,\ Q_e^1\ \wedge\ Q_e^2\ \right\}},\ l_{start}, l_{next}, l_{retry}, l_{exc}\right) =$$

$$\nabla_S\left(\frac{\{P_1\}\ s_1\ \{Q_1\} \quad \{P_2\}\ s_1\ \{Q_2\}}{\{P_1\ \wedge\ P_2\}\ s_1\ \{Q_1\ \wedge\ Q_2\}}\ ,\ l_{start}, l_{next}, l_{retry}, l_{exc}\right) =$$

We create the two proofs

$$\nabla_S\left(\ T_a,\ l_{start}, l_b, l_{retry}, l_{exc}\right)$$

and

$$\nabla_S\left(\ T_b,\ l_b, l_{next}, l_{retry}, l_{exc}\right)$$

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:

$$\begin{array}{ll} \{ \ 0 \leq \text{i} < \text{n} \ \} & IL\_0000 : \mathsf{nop} \\ \{ \ (0 \leq \text{i} < \text{n}) \wedge \text{y=y}\} & IL\_0001 : \mathsf{ldloc\ i} \end{array}$$

by the following bytecode proof without nop instructions:

$$\{ \ 0 \leq \text{i} < \text{n} \ \} \quad 00 : \mathsf{ldloc\ i}$$

6.5 Translation of assignment, conditional and compositional instructions

Assignment instruction

In the assignment translation, first the expression is translated using ∇_E. Then, the result is stored to x using stloc. The definition of the translation is the following:

$$\nabla_S \left(\frac{}{\left\{ \ P[e/x] \ \right\} \ \ x := e \ \ \left\{ \ P \ , \ false \ , \ false \ \right\}} , \ l_{start}, l_{next}, l_{retry}, l_{exc} \right) =$$

$$\begin{array}{ll} & \nabla_E \ (\ P[e/x] \ , \ \ e \ \ , \ (shift(P[e/x]) \ \wedge \ s(0) = e) \ , \ l_{start}) \\ \left\{ \ shift(P[e/x]) \wedge \ s(0) = e \ \right\} & l_b : \mathsf{stloc} \ x \end{array}$$

Conditional instruction

In this translation, the expression e is translated using ∇_E. If e is true (e is on the top of the stack), control is transferred to l_e and the translation of s_1 is obtained using ∇_S. Otherwise, s_2 is translated and control is transferred to the end (l_{next}).

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

$$T_{S_1} \equiv \frac{Tree_1}{\left\{ \ P \ \wedge \ e \ \right\} \ \ s_1 \ \ \left\{ \ Q_n \ , \ Q_r \ , \ Q_e \ \right\}}$$

$$T_{S_2} \equiv \frac{\mathit{Tree}_2}{\left\{\ P \ \wedge\ \neg e\ \right\}\ s_2\ \left\{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \right\}}$$

The translation is defined as follows:

$$\nabla_S \left(\frac{T_{S_1} \qquad T_{S_2}}{\left\{\ P\ \right\}\ \begin{array}{l}\texttt{if } e \texttt{ then}\\ \quad s_1\\ \texttt{else}\\ \quad s_2\\ \texttt{end}\end{array}\ \left\{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \right\}},\ l_{start}, l_{next}, l_{retry}, l_{exc} \right) =$$

$$\begin{array}{ll}
 & \nabla_E\ (\ P,\ \ e\ \ ,\ (\mathit{shift}(P)\ \wedge\ s(0) = e)\ ,\ l_{start})\\
\{\mathit{shift}(P) \wedge s(0) = e\} & l_b : \textsf{brtrue}\ l_e\\
 & \nabla_S\ \ (T_{S_2},\ l_c, l_d, l_{retry}, l_{exc})\\
\{Q_n\} & l_d : \textsf{br}\ l_{next}\\
 & \nabla_S\ \ (T_{S_1},\ l_e, l_{next}, l_{retry}, l_{exc})
\end{array}$$

Compositional instruction

Compositional instructions are the simplest instructions to translate. The translation of s_2 is added after the translation of s_1 where the starting label is updated to l_b.

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

$$T_{S_1} \equiv \frac{\mathit{Tree}_1}{\left\{\ P\ \right\}\ s_1\ \left\{\ Q_n\ ,\ R_r\ ,\ R_e\ \right\}}$$

$$T_{S_2} \equiv \frac{\mathit{Tree}_2}{\left\{\ Q_n\ \right\}\ s_2\ \left\{\ R_n\ ,\ R_r\ ,\ R_e\ \right\}}$$

The definition of the translation is the following:

$$\nabla_S \left(\frac{T_{S_1} \qquad T_{S_2}}{\left\{\ P\ \right\}\ s_1; s_2\ \left\{\ R_n\ ,\ R_r\ ,\ R_e\ \right\}},\ l_{start}, l_{next}, l_{retry}, l_{exc} \right) =$$

$$\begin{array}{l}
\nabla_S\,(T_{S_1},\ l_{start}, l_b, l_{retry}, l_{exc})\\
\nabla_S\,(T_{S_2},\ l_b, l_{next}, l_{retry}, l_{exc})
\end{array}$$

6.6 Creation Instruction, read and write attribute, and routine invocation translation

In the following we present the translation for creation instruction, read and write attribute and routine invocation.

6.6.1 Creation Instruction

$$\nabla_S \left(\frac{\{\ P\ \}\ \ T@make(p)\ \ \{\ Q_n\ ,\ Q_e\ \}}{\left\{ P \begin{bmatrix} new(\$, T)/Current, \\ \$ < T > /\$, \\ e/p \end{bmatrix} \right\}\ x := \texttt{create}\ \{T\}.make(e) \left\{ \begin{array}{l} Q_n[x/Current]\ , \\ false\ , \\ Q_e[x/Current] \end{array} \right\}},\ l_{start}, l_{next}, l_{retry}, l_{exc} \right) =$$

$$\begin{array}{ll}
\{P[new(\$, T)/Current, \$ < T > /\$, e/p]\} & l_a : \text{newobj } root\_cluster.Impl.T \\
\{(shift(P)[s(0)/Current, e/p])\} & l_b : \text{stloc } x \\
\{x \neq Void\ \wedge\ P[x/Current, e/p]\} & l_c : \text{ldloc } x \\
\nabla_E\ (\ \{x \neq Void\ \wedge\ shift(P[x/Current, e/p])\ \wedge\ s(0) = x\}, e, & \\
\{x \neq Void\ \wedge\ shift^2(P[x/Current, e/p])\ \wedge\ s(1) = x\ \wedge\ s(0) = e\}, l_d) & \\
\{x \neq Void\ \wedge\ shift^2(P[x/Current, e/p])\ \wedge\ s(1) = x\ \wedge\ s(0) = e\} & l_e : \text{callvirt } T@make
\end{array}$$

6.6.2 Read Attribute

The translation of read attribute is done invoking the method *name*() where *name* is the name of the attribute. This method returns the field. Let S be the following precondition:

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

The translation is defined as follows:

$$\nabla_S \left(\frac{}{\{\ S\ \}\ \ x := y.S@a\ \ \{\ P\ ,\ false\ ,\ Q_e\ \}},\ l_{start}, l_{next}, l_{retry}, l_{exc} \right) =$$

$$\begin{array}{ll}
\{y \neq Void\ \wedge\ P[\$(iv(y, S@a))/x]\} & l_a : \text{ldloc } y \\
\{s(0) = y\ \wedge Shift(P[\$(iv(y, S@a))/x])\} & l_b : \text{callvirt } S@a() \\
\{s(0) = \$(iv(y, S@a))\ \wedge Shift(P[\$(iv(y, S@a))/x])\} & l_c : \text{stloc } x
\end{array}$$

6.6.3 Write Attribute

To write an attribute, the method *set_name* is used (where *name* is the name of the attribute to write). The object and the expression is pushed on the top of stack. Then, the method *set_name* is invoked. Let S be the following precondition:

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

The definition of the translation is the following:

$$\nabla_S \left(\frac{}{\{\ S\ \} \quad y.S@a := e \quad \{\ P\ ,\ \mathit{false}\ ,\ Q_e\ \}},\ l_{start}, l_{next}, l_{retry}, l_{exc} \right) =$$

$$\begin{array}{ll} \{y \neq \mathit{Void} \ \wedge \ P[\$ < iv(y, S@a) := e > /\$]\} & l_a : \mathsf{ldloc}\ y \\ \nabla_E(\{s(0) = y \ \wedge \ \mathit{shift}(P[\$ < iv(y, S@a) := e > /\$])\},\ e, & \\ \{s(1) = y \ \wedge \ s(0) = e \ \wedge \ \mathit{shift}^2(P[\$ < iv(y, S@a) := e > /\$])\}\ ,\ l_b) & \\ \{s(1) = y \ \wedge \ s(0) = e \ \wedge \ \mathit{shift}^2(P[\$ < iv(y, S@a) := e > /\$])\} & l_c : \mathsf{callvirt}\ S@set\_a \end{array}$$

6.6.4 Invocation

Let S be the following precondition:

$$S \equiv \left\{ \begin{array}{l} (y \neq \mathit{Void} \ \wedge \ P[y/\mathit{Current}, e/p]) \vee \\ (y = \mathit{Void} \ \wedge \ Q_e \left[\begin{array}{l} \$ < \mathit{NullPExc} > /\$, \\ \mathit{new}(\$, \mathit{NullPExc})/\mathit{excV} \end{array} \right]) \end{array} \right\}$$

The translation is defined as follows:

$$\nabla_S \left(\frac{\dfrac{\mathit{Tree}_1}{\left\{\ P\ \right\} \quad T : m(p) \quad \left\{\ Q_n\ ,\ \mathit{false}\ ,\ Q_e\ \right\}}}{\left\{\ S\ \right\} \quad x = y.T : m(e) \quad \left\{\ Q_n[x/\mathit{result}]\ ,\ \mathit{false}\ ,\ Q_e\ \right\}},\ l_{start}, l_{next}, l_{retry}, l_{exc} \right) =$$

$$\begin{array}{ll} \{y \neq \mathit{Void} \ \wedge \ P[y/\mathit{Current}, e/p]\} & l_a : \mathsf{ldloc}\ y \\ \nabla_E(\{\mathit{shift}(P[y/\mathit{Current}, e/p]) \ \wedge \ s(0) = y\},\ e, & \\ \{\mathit{shift}^2(P[y/\mathit{Current}, e/p]) \ \wedge \ s(1) = y \ \wedge \ s(0) = e\}\ ,\ l_b) & \\ \{\mathit{shift}^2(P[y/\mathit{Current}, e/p]) \ \wedge \ s(1) = y \ \wedge \ s(0) = e\} & l_c : \mathsf{callvirt}\ T : m \\ \{Q_n[s(0)/\mathit{result}]\} & l_d : \mathsf{stloc}\ x \end{array}$$

6.7 Translation of routines Rules

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

6.7.1 Routines implementation

$$\nabla_S\left(\frac{\dfrac{Tree_1}{\left\{\ P\ \right\}\ \ body(T@m)\ \left\{\ Q_n\ ,\ false\ ,\ Q_e\ \right\}}}{\left\{\ P\ \right\}\ \ T@m\ \ \left\{\ Q_n\ ,\ false\ ,\ Q_e\ \right\}},\ l_{start}, l_{next}, l_{retry}, l_{exc}\right) =$$

$$\nabla_S\left(\frac{Tree_1}{\left\{\ P\ \right\}\ \ body(T@m)\ \left\{\ Q_n\ ,\ false\ ,\ Q_e\ \right\}},\ l_{start}, l_b, l_{retry}, l_{exc}\right)$$
$\{\ Q_n\ \}\ \ l_b : \mathsf{ret}$

6.7.2 Local rule

$$\nabla_S\left(\frac{\dfrac{Tree_1}{\left\{\begin{array}{l} P \wedge v_1 = init(T_1) \\ \wedge \ldots \wedge v_n = init(T_n)\end{array}\right\}\ s\ \left\{\ Q_n\ ,\ false\ ,\ Q_e\ \right\}}}{\left\{\ P\ \right\}\ \ \texttt{local}\ \ T_1\ v_1;\ \ldots\ T_n\ v_n;\ s\ \ \left\{\ Q_n\ ,\ false\ ,\ Q_e\ \right\}},\ l_{start}, l_{next}, l_{retry}, l_{exc}\right) =$$

$$\nabla_S\left(\frac{Tree_1}{\left\{\ P\ \right\}\ \ v_1 := init(T_1)\ \ \left\{\ P\ \wedge\ v_1 = init(T_1)\ ,\ false\ ,\ false\ \right\}},\ l_{start}, l_b, l_{retry}, l_{exc}\right)$$
...

$$\nabla_S\left(\frac{Tree_1}{\left\{\begin{array}{l} P\ \wedge \\ v_1 = init(T_1) \\ \wedge \ldots \wedge \\ v_{n-1} = init(T_{n-1})\end{array}\right\}\ v_n := init(T_n)\ \left\{\left(\begin{array}{l} P\ \wedge \\ v_1 = init(T_1) \\ \wedge \ldots \wedge \\ v_n = init(T_n)\end{array}\right),\ false\ ,\ false\right\}},\ l_b, l_c, l_{retry}, l_{exc}\right)$$

$$\nabla_S\left(\frac{Tree_1}{\left\{\ P\ \wedge\ v_1 = init(T_1)\ \wedge\ \ldots \wedge\ v_n = init(T_n)\ \right\}\ \ s\ \ \left\{\ Q_n\ ,\ false\ ,\ Q_e\ \right\}},\ l_c, l_{next}, l_c, l_{exc}\right)$$

where P does not contain $v_1, ..., v_n$.

6.8 Translation of instructions specific of Eiffel

In this section, we present the translation of instructions specific of Eiffel. In Subsection 6.8.1 we describe the translation of Eiffel loops. `rescue` and `retry` instructions are translated in Subsection 6.8.3 and 6.8.2. Translation of `check` and `debug` instructions are presented in Subsections 6.8.4 and 6.8.5 resp. Finally, once procedures and once functions are translated in Subsections 6.8.6 and 6.8.7 resp.

6.8.1 Loop Instruction translation

In the loop translation, first s_1 is translated using ∇_S. In this translation, the next label is updated to l_b and the remaining labels keep unchanged. Then, control is transferred l_d and the loop expression is evaluated. If its value is false, control is transferred to l_c where the body of the loop (s_2) is translated. The definition of the translation is the following:

$$\nabla_S \left(\frac{\begin{array}{c} \left\{ \; P \; \right\} \; s_1 \; \left\{ \; I_n \; , \; Q_r \; , \; R_e \; \right\} \\ \left\{ \; \neg e \; \wedge \; I_n \; \right\} \; s_2 \; \left\{ \; I_n \; , \; Q_r \; , \; R_e \; \right\} \end{array}}{\left\{ \; P \; \right\} \; \begin{array}{l} \texttt{from } s_1 \texttt{ invariant } I_n \\ \texttt{until } e \\ \texttt{loop } s_2 \texttt{ end} \end{array} \; \left\{ \; (I_n \; \wedge \; e) \; , \; Q_r \; , \; R_e \; \right\}}, \; l_{start}, l_{next}, l_{retry}, l_{exc} \right) =$$

$$\begin{array}{ll} & \nabla_S \left(\left\{ \; P \; \right\} \; s_1 \; \left\{ \; I_n \; , \; Q_r \; , \; R_e \; \right\}, \; l_{start}, l_b, l_{retry}, l_{exc} \right) \\ \{I_n\} & l_b : \mathsf{br} \; l_d \\ & \nabla_S \left(\left\{ \; \neg e \wedge I_n \; \right\} \; s_2 \; \left\{ \; I_n \; , \; Q_r \; , \; R_e \; \right\}, \; l_c, l_d, l_{retry}, l_{exc} \right) \\ & \nabla_E(\; I_n, \; e, \; \{shift(I_n) \; \wedge \; s(0) = e\}, \; l_d \;) \\ \{shift(I_n) \; \wedge \; s(0) = e\} & l_e : \mathsf{brfalse} \; l_c \end{array}$$

6.8.2 Retry

The `retry` instruction is translated using an unconditional jump to the beginning of the routine (which is store in the label l_{retry}). The translation is defined as follows:

$$\nabla_S \left(\frac{}{\left\{ \; P \; \right\} \; \texttt{retry} \; \left\{ \; false \; , \; P \; , \; false \; \right\}}, \; l_{start}, l_{next}, l_{retry}, l_{exc} \right) =$$

$$\{ \; P \; \} \quad l_a : \mathsf{br} \; l_{retry}$$

6.8.3 Rescue

The `rescue` rule is translated using .try and catch CIL instructions. First, the instruction s_1 is translated in a .try block. The exception label is updated to l_c because if an exception occurs in s_1, control will be transferred to the catch block at l_c. Then, the instruction s_2 is translated in a catch block. To do that, first, the exception object is stored in a temporary variable and then s_2 is translated. In this translation, the `retry` label is updated to l_a (the beginning of the routine). Finally, we the exception is pushed on top of the stack and it is re-thrown (in case s_2 terminates normally).

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

$$T_{S_1} \equiv \frac{Tree_1}{\left\{ \; I_n \; \right\} \; s_1 \; \left\{ \; Q_n \; , \; false \; , \; I'_n \; \right\}}$$

$$T_{S_2} \equiv \frac{Tree_2}{\left\{ \; I'_n \; \right\} \; s_2 \; \left\{ \; R_e \; , \; I_n \; , \; R_e \; \right\}}$$

The translation is defined as follows:

$$\nabla_S\left(\frac{T_{S_1}\qquad T_{S_2}}{\left\{\ P\ \right\}\ s_1\ \texttt{rescue}\ s_2\ \left\{\ Q_n\ ,\ \mathit{false}\ ,\ R_e\ \right\}},\ l_{start}, l_{next}, l_{retry}, l_{exc}\right) =$$

$$\begin{array}{ll}
 & \mathsf{.try}\{ \\
 & \quad \nabla_S\ \left(T_{S_1}, l_{start}, l_b, l_{retry}, l_c\ \right) \\
\{Q_n\} & \quad l_b : \mathsf{leave}\ l_{next} \\
 & \} \\
 & \mathsf{catch}\ [\mathit{mscorlib}]\ \mathit{System.Exception}\ \{ \\
\{I'_n\ \wedge\ \mathit{excV} \neq \mathit{null}\ \wedge\ s(0) = \mathit{excV}\} & \quad l_c :\ \mathsf{stloc}\ \mathit{last\_exception} \\
\{I'_n\ \} & \quad \nabla_S\ \left(T_{S_2}, l_d, l_e, l_a, l_{exc}\ \right) \\
\{R_e\ \} & \quad l_e :\ \mathsf{ldloc}\ \mathit{last\_exception} \\
\{R_e\ \wedge\ s(0) = \mathit{last\_exception}\} & \quad l_f : \mathsf{rethrow} \\
 & \}
\end{array}$$

6.8.4 Check instruction translation

When a `check` instruction is translated, first the expression e is pushed on top of the stack. If e evaluates to true, control is transferred to the next instruction. Otherwise, an exception is thrown putting a new exception object on the top of the stack and then using the throw instruction. The definition of the translation is the following:

$$\nabla_S\left(\frac{}{\left\{\ P\ \right\}\ \texttt{check}\ e\ \texttt{end}\ \left\{\ (P\ \wedge\ e\ )\ ,\ \mathit{false}\ ,\ (P\ \wedge\ \neg e\ )\ \right\}},\ l_{start}, l_{next}, l_{retry}, l_{exc}\right) =$$

$$\begin{array}{ll}
 & \nabla_E(\ P,\ e,\ \{\mathit{shift}(P)\ \wedge\ s(0) = e\},\ l_a\ ) \\
\{\mathit{shift}(P)\ \wedge\ s(0) = e\} & l_b : \mathsf{brtrue}\ l_{next} \\
\{P\ \wedge\ \neg e\} & l_c : \mathsf{newobj}\ \mathit{Exception}() \\
\{P\ \wedge\ \neg e \wedge s(0) = \mathit{new}(\$, \mathit{Exception})\} & l_d : \mathsf{throw}
\end{array}$$

6.8.5 Debug instruction translation

To translate the debug instruction, we need to check the value of $\mathcal{D}$. $\mathcal{D}$ is a static variable, so there is not need to generate code that checks the value of $\mathcal{D}$ and according to it, executes s_1 or not. Thus, $\mathcal{D}$ is checked in compile time. If $\mathcal{D} = \mathit{debug}$, the CIL code and proof for s_1 are generated (using ∇_S). Otherwise, a nop instruction is generated. The translation is defined as follows:

$$\nabla_S\left(\frac{\dfrac{\mathit{Tree}_1}{\left\{\ P\ \right\}\ s_1\ \left\{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \right\}}}{\left\{\begin{array}{l}(P\ \wedge\ \mathcal{D} = \mathit{debug})\ \vee \\ (P'\ \wedge\ \mathcal{D} = \mathit{not\_debug})\end{array}\right\}\ \texttt{debug}\ s_1\ \texttt{end}\ \left\{\ Q_n\ \vee\ P'\ ,\ Q_r\ ,\ Q_e\ \right\}},\ l_{start}, l_{next}, l_{retry}, l_{exc}\right) =$$

$\texttt{if}\ \mathcal{D} = \mathit{debug}\ \texttt{then}$

$$\nabla_S\left(\frac{\mathit{Tree}_1}{\left\{\ P\ \right\}\ s_1\ \left\{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \right\}},\ l_{start}, l_{next}, l_{retry}, l_{exc}\right)$$

`else`

$\{P'\}\ l_a : \mathsf{nop}$

6.8.6 Once procedures translation

In this translation, first the variable C_p_done is evaluated. If its value is true, control is transferred to the end of the procedure (l_i) and if the variable $C\_p\_exception$ is null the procedure terminates normally; otherwise the $C\_p\_exception$ is thrown. If C_p_done is false, C_p_done is set to true and the body of the procedure is executed. If an exception is throw in the body of the procedure, the exception is store in $C\_p\_exception$ to be able to reproduce the same exception in later executions of the procedure.

Let S be the following precondition:

$$S \equiv \left\{ \begin{array}{l} (\neg C\_p\_done \ \wedge \ P) \vee \\ \left(\begin{array}{l} C\_p\_done \ \wedge \ P' \ \wedge \\ C\_p\_exception = Void \end{array} \right) \vee \\ (C\_p\_done \ \wedge \ P'' \ \wedge \ C\_p\_exception \neq Void) \end{array} \right\}$$

and Q'_n and Q'_e be the following postconditions:

$$Q'_n \equiv \left\{ \begin{array}{l} C\_p\_done \ \wedge \ C\_p\_exception = Void \ \wedge \\ (Q_n \vee P') \end{array} \right\}$$

$$Q'_e \equiv \left\{ \begin{array}{l} C\_p\_done \ \wedge \ C\_p\_exception \neq Void \ \wedge \\ (Q_e \ \vee P'') \end{array} \right\}$$

Let T_{body} be the following proof tree:

$$T_{body} \equiv \frac{Tree_1}{\left\{ \begin{array}{l} P[false/C\_p\_done] \ \wedge \\ C\_p\_done \end{array} \right\} \ body(C:p) \ \left\{ \begin{array}{l} \left(\ Q_n \ \wedge \ C\_p\_done \ \right) \ , \\ \\ false \ , \\ \\ \left(\ Q_e \ \wedge \ C\_p\_done \ \right) \end{array} \right\}}$$

The definition of the translation is the following:

$$\nabla_S \left(\frac{T_{body}}{\left\{ \ S \ \right\} \ \ C:p(i) \ \left\{ \ Q'_n \ , \ false \ , \ Q'_e \ \right\}} , \ l_{start}, l_{next}, l_{retry}, l_{exc} \right) =$$

$$
\begin{array}{ll}
\{S\} & l_a : \mathsf{ldsfld}\ \mathit{C\_p\_done} \\
\{S\ \wedge\ s(0) = \mathit{C\_p\_done}\} & l_b : \mathsf{brtrue}\ l_i \\
\{P\ \wedge\ \neg \mathit{C\_p\_done}\} & l_c : \mathsf{ldc}\ \mathit{true} \\
\{P\ \wedge\ \neg \mathit{C\_p\_done}\ \wedge\ s(0) = \mathit{true}\} & l_d : \mathsf{stsfld}\ \mathit{C\_p\_done} \\
\mathit{.try} & \\
\{ & \\
\quad \nabla_S\ (T_{body}, l_e, l_f, l_{retry}, l_{exc}\ ) & \\
\quad \{Q_n\ \wedge\ \mathit{C\_p\_done}\} & l_f : \mathsf{leave}\ l_i \\
\} & \\
\mathit{catch}\ \ [\mathit{mscorlib}]\mathit{System.Exception} & \\
\{ & \\
\quad \{\mathit{shift}(Q_e)\ \wedge\ \mathit{excV} \neq \mathit{null}\ \wedge\ \mathit{C\_p\_done}\} & l_g : \mathsf{stsfld}\ \mathit{C\_p\_exception} \\
\quad \{Q_e\ \wedge\ \mathit{C\_p\_exception} \neq \mathit{null}\ \wedge\ \mathit{C\_p\_done}\} & l_h : \mathsf{rethrow} \\
\} & \\
\{Q'\} & l_i : \mathsf{ldsfld}\ \mathit{C\_p\_exception} \\
\{\mathit{shift}(Q')\ \wedge\ s(0) = \mathit{C\_p\_exception}\} & l_j : \mathsf{brfalse}\ l_m \\
\{\mathit{C\_p\_done}\ \wedge\ \mathit{C\_p\_exception} \neq \mathit{null}\ \wedge\ P''\ \} & l_k : \mathsf{ldsfld}\ \mathit{C\_p\_exception} \\
\left\{ \begin{array}{l} \mathit{C\_p\_done}\ \wedge\ \mathit{C\_p\_exception} \neq \mathit{null}\ \wedge\ P''\ \wedge \\ s(0) = \mathit{C\_p\_exception} \end{array} \right\} & l_l : \mathsf{throw} \\
\left\{ \begin{array}{l} \mathit{C\_p\_done}\ \wedge\ \mathit{C\_p\_exception} = \mathit{null}\ \wedge \\ \left(\ Q_n\ \vee P'\ \right) \wedge s(0) = \mathit{C\_p\_result} \end{array} \right\} & l_m : \mathsf{ret}
\end{array}
$$

$$
\mathit{where}\ Q' \equiv \left\{ \begin{array}{l} \mathit{C\_p\_done}\ \wedge \\ \left(\begin{array}{l} (Q_n\ \vee\ P'\ \vee \\ (P''\ \wedge\ \mathit{C\_p\_exception} \neq \mathit{null}) \end{array} \right) \end{array} \right\}
$$

6.8.7 Once functions translation

The translation of once functions is similar to once routines' translation. The only difference is that the result of the first invocation of the function is stored in $\mathit{C\_f\_result}$. Later invocations returns the value stored in $\mathit{C\_f\_result}$.

Let S be the following precondition

$$
S \equiv \left\{ \begin{array}{l} (\neg \mathit{C\_f\_done}\ \wedge\ P) \vee \\ \left(\begin{array}{l} \mathit{C\_f\_done}\ \wedge\ P'\ \wedge \\ \mathit{C\_f\_result} = \mathit{C\_F\_RESULT}\ \wedge \\ \mathit{C\_f\_exception} = \mathit{Void} \end{array} \right) \vee \\ (\mathit{C\_f\_done}\ \wedge\ P''\ \wedge\ \mathit{C\_f\_exception} \neq \mathit{Void}) \end{array} \right\}
$$

and Q'_n and Q'_e be the following postconditions:

$$Q'_n \equiv \left\{ \begin{array}{l} C\_f\_done \;\wedge\; C\_f\_exception = Void \wedge \\ (Q_n \vee \left(\begin{array}{l} P' \;\wedge\; result = C\_F\_RESULT \wedge \\ C\_f\_result = C\_F\_RESULT \end{array} \right) \end{array} \right\}$$

$$Q'_e \equiv \left\{ \begin{array}{l} C\_f\_done \;\wedge\; C\_f\_exception \neq Void \wedge \\ (Q_e \;\vee P'') \end{array} \right\}$$

Let T_{body} be the following proof tree:

$$T_{body} \equiv \frac{Tree_1}{\left\{ \begin{array}{l} P[false/C\_f\_done] \wedge \\ C\_f\_done \end{array} \right\} \; body(C:f) \; \left\{ \begin{array}{l} \left(\; Q_n \;\wedge\; C\_f\_done \; \right) \; , \\ \\ false \; , \\ \\ \left(\; Q_e \;\wedge\; C\_f\_done \; \right) \end{array} \right\}}$$

The translation is defined as follows:

$$\nabla_S \left(\frac{T_{body}}{\left\{ \; S \; \right\} \quad C:f(i) \quad \left\{ \; Q'_n \; , \; false \; , \; Q'_e \; \right\}} , \; l_{start}, l_{next}, l_{retry}, l_{exc} \right) =$$

$\{S\}$	l_a : ldsfld $\mathit{C\_f\_done}$
$\{S \ \wedge\ s(0) = \mathit{C\_f\_done}\}$	l_b : brtrue l_k
$\{P \ \wedge\ \neg \mathit{C\_f\_done}\}$	l_c : ldc true
$\{P \ \wedge\ \neg \mathit{C\_f\_done} \ \wedge\ s(0) = \mathit{true}\}$	l_d : stsfld $\mathit{C\_f\_done}$
$.\mathit{try}$	
$\{$	
$\quad\nabla_S \ (T_{body}, l_e, l_f, l_{retry}, l_g\ )$	
$\quad\{Q_n \ \wedge\ \mathit{C\_f\_done}\}$	l_f : ldloc result
$\quad\{Q_n \ \wedge\ \mathit{C\_f\_done} \ \wedge\ s(0) = \mathit{result}\}$	l_g : stsfld $\mathit{C\_f\_result}$
$\quad\{Q_n \ \wedge\ \mathit{C\_f\_done} \ \wedge\ \mathit{result} = \mathit{C\_f\_result}\}$	l_h : leave l_k
$\}$	
$\mathit{catch}\ \ [\mathit{mscorlib}]\mathit{System.Object}$	
$\{$	
$\quad\{\mathit{shift}(Q_e) \ \wedge\ \mathit{excV} \neq \mathit{null} \ \wedge\ \mathit{C\_f\_done}\}$	l_i : stsfld $\mathit{C\_f\_exception}$
$\quad\{Q_e \ \wedge\ \mathit{C\_f\_exception} \neq \mathit{null} \ \wedge\ \mathit{C\_f\_done}\}$	l_j : rethrow
$\}$	
$\{Q'\}$	l_k : ldsfld $\mathit{C\_f\_exception}$
$\{\mathit{shift}(Q') \ \wedge\ s(0) = \mathit{C\_f\_exception}\}$	l_l : brfalse l_o
$\{\mathit{C\_f\_done} \ \wedge\ \mathit{C\_f\_exception} \neq \mathit{null} \ \wedge\ P''\ \}$	l_m : ldsfld $\mathit{C\_f\_exception}$
$\left\{\begin{array}{l} \mathit{C\_f\_done} \ \wedge\ \mathit{C\_f\_exception} \neq \mathit{null} \ \wedge\ P'' \ \wedge \\ s(0) = \mathit{C\_f\_exception} \end{array}\right\}$	l_n : throw
$\left\{\begin{array}{l} \mathit{C\_f\_done} \ \wedge\ \mathit{C\_f\_exception} = \mathit{null} \ \wedge \\ \left(Q_n \ \vee \left(\begin{array}{l} P' \ \wedge\ \mathit{result} = \mathit{C\_F\_RESULT} \ \wedge \\ \mathit{C\_f\_result} = \mathit{C\_F\_RESULT} \end{array} \right) \right) \end{array}\right\}$	l_o : ldsfld $\mathit{C\_f\_result}$
$\left\{\begin{array}{l} \mathit{C\_f\_done} \ \wedge\ \mathit{C\_f\_exception} = \mathit{null} \ \wedge \\ \left(Q_n \ \vee \left(\begin{array}{l} P' \ \wedge\ \mathit{result} = \mathit{C\_F\_RESULT} \ \wedge \\ \mathit{C\_f\_result} = \mathit{C\_F\_RESULT} \end{array} \right) \right) \ \wedge \\ s(0) = \mathit{C\_f\_result} \end{array}\right\}$	l_p : ret

$$\mathit{where}\ \ Q' \equiv \left\{\begin{array}{l} \mathit{C\_f\_done} \ \wedge \\ \left(\begin{array}{l} (Q_n \ \vee \\ \left(\begin{array}{l} P' \ \wedge\ \mathit{C\_f\_result} = \mathit{C\_F\_RESULT} \ \wedge \\ \mathit{C\_f\_exception} = \mathit{null} \end{array} \right) \ \vee \\ (P'' \ \wedge\ \mathit{C\_f\_exception} \neq \mathit{null}) \end{array} \right) \end{array}\right\}$$

7 Specification Translation

To be able to define the specification translation, we need to know the structure of the formula we are translating. Thus, we have defined a deep embedding of the Eiffel contract language. Then, we have defined translation functions to FOL. The datatype definitions, the translation functions and their soundness proof are formalized in Isabelle.

7.1 Datatype definitions

Eiffel contract are expressed using boolean expressions. Boolean expression are the logical operator ¬, ∧, ∨, *AndThen*, *OrElse*, *Xor*, *implies*, expressions equality, <, >, ≤, ≥ or type functions. Expressions are constants, local variables and parameters, attributes, routine calls, creation expressions, old expressions, boolean expressions or *Void*. The type functions are *ConformsTo* or *IsEqual*. We assume routines have only one argument.

datatype *EiffelContract* = **Requires_ensures** *boolExpr*

datatype *boolExpr* = **Const** *bool*
| **Neg** *boolExpr*
| **And** *boolExpr boolExpr*
| **Or** *boolExpr boolExpr*
| **AndThen** *boolExpr boolExpr*
| **OrElse** *boolExpr boolExpr*
| **Xor** *boolExpr boolExpr*
| **Impl** *boolExpr boolExpr*
| **Eq** *expr expr*
| **NotEq** *expr expr*
| **Less** *expr expr*
| **Greater** *expr expr*
| **LessE** *expr expr*
| **GreaterE** *expr expr*
| **Type** *typeFunc*

datatype *typeFunc* = **ConformsTo** *typeExpr typeExpr*
| **IsEqual** *typeExpr typeExpr*
| **IsNotEqual** *typeExpr typeExpr*

datatype *typeExpr* = **EType** *EiffelType*
| **Type** *expr*

datatype *expr* = **ConstInt** *int*
| **RefVar** *varID*
| **Att** *objID attribID*
| **CallR** *callRoutine*
| **Create** *EiffelType routine argument*
| **Old** *expr*
| **Bool** *boolExpr*
| **Void**

datatype *callRoutine* = **Call** *expr routine argument*

datatype *argument* = **Argument** *expr*

EiffelTypes are *Boolean*, *Integer*, classes with a *classID* or *None*. The notation $(cID : classID)$ means, given an Eiffel class c, cID(c) returns its *classID*.

datatype *EiffelType* = **Boolean**
| **Integer**
| **EClass** $(cID : classID)$
| **None**

Variables are local variables or parameters, Result, or Current:

datatype *var* = **Var** *vID EiffelType*
| **Result** *EiffelType*
| **Current** *EiffelType*

Attributes are defined with a variable *ID* and an *EiffelType*. Routines can take only one argument. Routines are defined with a routine *ID*, the argument type and the return type.

datatype *attrib* = **Attr** $(aID : attribID)$ *EiffelType*

datatype *routine* = **Routine** *routineID EiffelType EiffelType*

7.2 Object store and values

Objects stores are modelled by an abstract data type *store*. We use the object store presented in [15]. The Eiffel object store and the CIL object store are the same. The object store contains the following operations: $accessC(os, l)$ denotes reading the location l in store os; $alive(o, os)$ yields true if and only if object o is allocated in os; $new(os, C)$ returns a reference to a new object in the store os of type C; $alloc(os, C)$ denotes the store after allocating the object store $new(os, C)$; $update(os, l, v)$ updates the object store os at the location l with the value v:

$accessC$ ::	$store \Rightarrow location \Rightarrow value$
$alive$ ::	$value \Rightarrow store \Rightarrow bool$
$alloc$ ::	$store \Rightarrow classID \Rightarrow store$
new ::	$store \Rightarrow classID \Rightarrow value$
$update$ ::	$store \Rightarrow location \Rightarrow value \Rightarrow store$

Values are booleans, integers, the void value or references to objects. Objects are characterized by its class and an identifier of infinite sort *objID*.

datatype *value* = **BoolV** *bool*
| **IntV** *int*
| **ObjV** *classID objID*
| **VoidV**

7.3 Mapping Eiffel types to CIL

To define the translation from Eiffel contracts to *FOL*, we first define CIL types and mapping functions that map Eiffel types to CIL. CIL types are boolean, integer, inferfaces, classes and the null type.

datatype *CilType* = **BooleanCIL**
| **IntegerCIL**
| **Interface** *classID*
| **CilClass** *classID*
| **NullT**

We have defined two functions that map Eiffel types to CIL: (1) ∇_{int} maps an Eiffel type to a CIL interface; (2) ∇_{class} maps the type to a CIL implementation class. The functions are defined as follows:

$\nabla_{\mathbf{int}}$ :: *EiffelType* $\Rightarrow$ *CilType*		$\nabla_{\mathbf{class}}$ :: *EiffelType* $\Rightarrow$ *CilType*	
$\nabla_{\mathbf{int}}(Boolean)$	= **BooleanCIL**	$\nabla_{\mathbf{class}}(Boolean)$	= **BooleanCIL**
$\nabla_{\mathbf{int}}(Integer)$	= **IntegerCIL**	$\nabla_{\mathbf{class}}(Integer)$	= **IntegerCIL**
$\nabla_{\mathbf{int}}(EClass\ n)$	= **Interface** n	$\nabla_{\mathbf{class}}(EClass\ n)$	= **CilClass** n
$\nabla_{\mathbf{int}}(None)$	= **NullT**	$\nabla_{\mathbf{class}}(None)$	= **NullT**

To translate routine calls, we define method signatures in CIL and a translation function that maps Eiffel routines to CIL methods. The types t_1 and t_2 are mapped to CIL types using the function ∇_{int}.

datatype *methodIDcil* = **Method** *nat CilType CilType*

$\nabla_{\mathbf{r}}$ :: *routine* $\Rightarrow$ *methodCIL*

$\nabla_{\mathbf{r}}(Routine\ n\ t1\ t2) = (Method\ \ n\ \ (\nabla_{int}\ t1)\ \ (\nabla_{int}\ t2))$

7.4 Contract translation

The specification translation is performed using five translation functions: (1) ∇_b takes a boolean expression and returns a function that takes two stores and a state an returns a value; (2) ∇_{exp} translates expressions; (3) ∇_t translates type functions (conforms to and is equal); (4) ∇_{call} translates a routine call; and (5) ∇_{arg} translates arguments. *state* is a mapping from variables to values ($var \Rightarrow value$). The signatures of the translation functions are the following:

∇_b :: *boolExpr* $\Rightarrow$ (*store* $\Rightarrow$ *store* $\Rightarrow$ *state* $\Rightarrow$ *value*)
∇_{exp} :: *expr* $\Rightarrow$ (*store* $\Rightarrow$ *store* $\Rightarrow$ *state* $\Rightarrow$ *value*)
∇_t :: *typeFunc* $\Rightarrow$ (*store* $\Rightarrow$ *store* $\Rightarrow$ *state* $\Rightarrow$ *value*)
∇_{call} :: *callRoutine* $\Rightarrow$ (*store* $\Rightarrow$ *store* $\Rightarrow$ *state* $\Rightarrow$ *value*)
∇_{arg} :: *argument* $\Rightarrow$ (*store* $\Rightarrow$ *store* $\Rightarrow$ *state* $\Rightarrow$ *value*)
The definition of the function ∇_b is the following:

$$\begin{array}{ll}
\nabla_b(\textbf{Const}\ b) & = \lambda(h_1, h_2 :: store)\ (s :: state) : \\
 & \quad (BoolV\ \ b) \\
\nabla_b(\textbf{Neg}\ b) & = \lambda(h_1, h_2 :: store)\ (s :: state) : \\
 & \quad (BoolV \neg(aB(\nabla_b\ b\ h_1\ h_2\ s))) \\
\nabla_b(\textbf{And}\ b_1\ b_2) & = \lambda(h_1, h_2 :: store)\ (s :: state) : \\
 & \quad (BoolV (aB(\nabla_b\ b_1\ h_1\ h_2\ s)) \wedge (aB(\nabla_b\ b_2\ h_1\ h_2\ s))) \\
\nabla_b(\textbf{Or}\ b_1\ b_2) & = \lambda(h_1, h_2 :: store)\ (s :: state) : \\
 & \quad (BoolV (aB(\nabla_b\ b_1\ h_1\ h_2\ s)) \vee (aB(\nabla_b\ b_2\ h_1\ h_2\ s))) \\
\nabla_b(\textbf{AndThen}\ b_1\ b_2) & = \lambda(h_1, h_2 :: store)\ (s :: state) : \\
 & \quad (BoolV (aB(\nabla_b\ b_1\ h_1\ h_2\ s)) \wedge (aB(\nabla_b\ b_2\ h_1\ h_2\ s))) \\
\nabla_b(\textbf{OrElse}\ b_1\ b_2) & = \lambda(h_1, h_2 :: store)\ (s :: state) : \\
 & \quad (BoolV (aB(\nabla_b\ b_1\ h_1\ h_2\ s)) \vee (aB(\nabla_b\ b_2\ h_1\ h_2\ s))) \\
\nabla_b(\textbf{Xor}\ b_1\ b_2) & = \lambda(h_1, h_2 :: store)\ (s :: state) : \\
 & \quad (BoolV (aB(\nabla_b\ b_1\ h_1\ h_2\ s)) \vee (aB(\nabla_b\ b_2\ h_1\ h_2\ s))) \\
\nabla_b(\textbf{Impl}\ b_1\ b_2) & = \lambda(h_1, h_2 :: store)\ (s :: state) : \\
 & \quad (BoolV (aB(\nabla_b\ b_1\ h_1\ h_2\ s)) \longrightarrow (aB(\nabla_p\ b_2\ h_1\ h_2\ s))) \\
\nabla_b(\textbf{Eq}\ e_1\ e_2) & = \lambda(h_1, h_2 :: store)\ (s :: state) : \\
 & \quad (BoolV (aI(\nabla_{exp}\ e_1\ h_1\ h_2\ s)) = (aI(\nabla_{exp}\ e_2\ h_1\ h_2\ s))) \\
\nabla_b(\textbf{NotEq}\ e_1\ e_2) & = \lambda(h_1, h_2 :: store)\ (s :: state) : \\
 & \quad (BoolV (aI(\nabla_{exp}\ e_1\ h_1\ h_2\ s)) \neq (aI(\nabla_{exp}\ e_2\ h_1\ h_2\ s))) \\
\nabla_b(\textbf{Less}\ e_1\ e_2) & = \lambda(h_1, h_2 :: store)\ (s :: state) : \\
 & \quad (BoolV (aI(\nabla_{exp}\ e_1\ h_1\ h_2\ s)) < (aI(\nabla_{exp}\ e_2\ h_1\ h_2\ s))) \\
\nabla_b(\textbf{Greater}\ e_1\ e_2) & = \lambda(h_1, h_2 :: store)\ (s :: state) : \\
 & \quad (BoolV (aI(\nabla_{exp}\ e_1\ h_1\ h_2\ s)) > (aI(\nabla_{exp}\ e_2\ h_1\ h_2\ s))) \\
\nabla_b(\textbf{LessE}\ e_1\ e_2) & = \lambda(h_1, h_2 :: store)\ (s :: state) : \\
 & \quad (BoolV (aI(\nabla_{exp}\ e_1\ h_1\ h_2\ s)) \leq (aI(\nabla_{exp}\ e_2\ h_1\ h_2\ s))) \\
\nabla_b(\textbf{GreaterE}\ e_1\ e_2) & = \lambda(h_1, h_2 :: store)\ (s :: state) : \\
 & \quad (BoolV (aI(\nabla_{exp}\ e_1\ h_1\ h_2\ s)) \geq (aI(\nabla_{exp}\ e_2\ h_1\ h_2\ s))) \\
\nabla_b(\textbf{Type}\ e) & = \lambda(h_1, h_2 :: store)\ (s :: state) : \\
 & \quad (\nabla_t\ e\ h_1\ h_2\ s))
\end{array}$$

The function ∇_t is defined as follows:

$$\begin{array}{ll}
\nabla_t(\textbf{ConformsTo}\ t_1\ t_2) = & \lambda(h_1, h_2 :: store)(s :: state) : \\
 & \quad (BoolV (\nabla_{int}(\nabla_{type} t_1)) \preceq_c (\nabla_{int}(\nabla_{type} t_2))) \\
\nabla_t(\textbf{IsEqual}\ t_1\ t_2) = & \lambda(h_1, h_2 :: store)(s :: state) : \\
 & \quad (BoolV (\nabla_{class}(\nabla_{type} t_1)) = (\nabla_{class}(\nabla_{type} t_2)))
\end{array}$$

The function ∇_{type} given a type expression returns its Eiffel type:

$$\begin{array}{ll}
\nabla_{\textbf{type}} ::\ typeExp \Rightarrow\ \ EiffelType & \\
\quad \nabla_{\textbf{type}}(EType\ t) & = t \\
\quad \nabla_{\textbf{type}}(Expression\ e) & = (typeOf\ \ e)
\end{array}$$

The definition of the function ∇_{exp} is the following:

$$
\begin{array}{ll}
\nabla_{exp}(\textbf{ConstInt}\ i) = & \lambda(h_1, h_2 :: store)(s :: state) : \\
& \quad (IntV\ \ i) \\
\nabla_{exp}(\textbf{RefVar}\ v) = & \lambda(h_1, h_2 :: store)(s :: state) : \\
& \quad (s(v)) \\
\nabla_{exp}(\textbf{Att}\ ob\ \ a) = & \lambda(h_1, h_2 :: store)(s :: state) : \\
& \quad (accessC\ \ h_1\ \ (Loc\ \ (get\_fieldID\ \ a)\ \ ob)) \\
\nabla_{exp}(\textbf{CallR}\ crt) = & \lambda(h_1, h_2 :: store)(s :: state) : \\
& \quad (\nabla_{call}\ crt\ \ h_1\ \ h2\ \ s) \\
\nabla_{exp}(\textbf{Precursor}\ t_1\ \ rt\ \ par) = & \lambda(h_1, h_2 :: store)(s :: state) : \\
& \quad (invokeValCIL\ \ h_1\ \ (\nabla_r\ \ rt)\ \ (s\ \ (Currentt_1))\ \ (\nabla_{arg}\ \ par\ \ h_1\ \ h_2\ \ s)) \\
\nabla_{exp}(\textbf{Create}\ t\ \ rt\ \ p) = & \lambda(h_1, h_2 :: store)(s :: state) : \\
& \quad (new\ \ (alloc\ \ h_1\ \ (get\_classID\ \ t))\ \ (get\_classID\ \ t)) \\
\nabla_{exp}(\textbf{Plus}\ e_1\ \ e_2) = & \lambda(h_1, h_2 :: store)(s :: state) : \\
& \quad (IntV((aI(\nabla_{exp}\ \ e_1\ \ h_1\ \ h_2\ \ s)) + (aI(\nabla_{exp}\ \ e_2\ \ h_1\ \ h_2\ \ s)))) \\
\nabla_{exp}(\textbf{Minus}\ e_1\ \ e_2) = & \lambda(h_1, h_2 :: store)(s :: state) : \\
& \quad (IntV((aI(\nabla_{exp}\ \ e_1\ \ h_1\ \ h_2\ \ s)) - (aI(\nabla_{exp}\ \ e_2\ \ h_1\ \ h_2\ \ s)))) \\
\nabla_{exp}(\textbf{Mul}\ e_1\ \ e_2) = & \lambda(h_1, h_2 :: store)(s :: state) : \\
& \quad (IntV((aI(\nabla_{exp}\ \ e_1\ \ h_1\ \ h_2\ \ s)) * (aI(\nabla_{exp}\ \ e_2\ \ h_1\ \ h_2\ \ s)))) \\
\nabla_{exp}(\textbf{Div}\ e_1\ \ e_2) = & \lambda(h_1, h_2 :: store)(s :: state) : \\
& \quad (IntV((aI(\nabla_{exp}\ \ e_1\ \ h_1\ \ h_2\ \ s))div(aI(\nabla_{exp}\ \ e_2\ \ h_1\ \ h_2\ \ s)))) \\
\nabla_{exp}(\textbf{Old}\ e) = & \lambda(h_1, h_2 :: store)(s :: state) : \\
& \quad (\nabla_{exp}\ \ e\ \ h_2\ \ h_2\ \ s) \\
\nabla_{exp}(\textbf{Bool}\ b) = & \lambda(h_1, h_2 :: store)(s :: state) : \\
& \quad (\nabla_b\ \ b\ \ h_1\ \ h_2\ \ s) \\
\nabla_{exp}(\textbf{Void}) = & \lambda(h_1, h_2 :: store)(s :: state) : \\
& \quad (VoidV)
\end{array}
$$

The function ∇_{call} is defined as follows:

$$
\begin{array}{l}
\nabla_{call}(\textbf{Call}\ e_1\ \ rt\ \ p) = \lambda(h_1, h_2 :: store)(s :: state) : \\
\qquad\qquad (invokeValCIL\ \ h_1\ \ (\nabla_r\ \ rt)\ \ (\nabla_{exp}\ \ e_1\ \ h_1\ \ h_2\ \ s)(\nabla_{arg}\ \ p\ \ h_1\ \ h_2\ \ s))
\end{array}
$$

The function *invokeValCIL* takes a CIL method m and two values (its parameter p and invoker e_1) and returns the value of the result of invoking the method m with the invoker e_1 and parameter p.

The definition of the function ∇_{arg} is the following:

$$
\begin{array}{ll}
\nabla_{arg}(\textbf{Argument}\ e) = & \lambda(h_1, h_2 :: store)(s :: state) : \\
& \quad (\nabla_{exp}\ \ e\ \ h_1\ \ h_2\ \ s)
\end{array}
$$

8 Example

8.1 Application of rescue and retry logic

Table 4 presents the bytecode proof generated by the translation functions from the source proof of figure 1.

Table 4: Final Bytecode proof of the source proof of table 1.

$\{op_2 \neq 0\}$	l_{01}: ldc 0
$\{op_2 \neq 0\ \wedge\ s(0) = 0\}$	l_{02}: stloc attempt
$\{op_2 \neq 0\ \wedge\ attempt = 0\}$	l_{03}: nop
	.try {

Continued on next page

$\left\{\left(op_2 \neq 0 \ \wedge \ \left(operation = 1 \ \Rightarrow \ \begin{array}{l} attempt \geq 0 \wedge \\ attempt \leq 3 \end{array}\right)\right) \equiv I\right\}$	l_{a01}: ldarg operation
$\{shift(I) \ \wedge \ s(0) = operation\}$	l_{a02}: ldc 1
$\{shift^2(I) \ \wedge \ s(1) = operation \wedge \ s(0) = 1\}$	l_{a03}: $binop_=$
$\{shift(I) \ \wedge \ s(0) = (operation = 1)\}$	l_{a04}: brfalse l_{a15}
$\{(op_2 \neq 0 \ \wedge \ attempt \geq 0 \ \wedge \ attempt \leq 3) \equiv Q\}$	l_{a05}: ldloc attempt
$\{shift(Q) \ \wedge \ s(0) = attempt\}$	l_{a06}: ldc 3
$\{shift^2(Q) \ \wedge \ s(1) = attempt \ \wedge \ s(0) = 3\}$	l_{a07}: $binop_<$
$\{shift(Q) \ \wedge \ s(0) = (attempt < 3)\}$	l_{a08}: brfalse l_{a10}
$\{(op_2 \neq 0 \ \wedge \ attempt \geq 0 \ \wedge \ attempt < 3) \equiv Q''\}$	l_{a09}: call open_a_file
$\{Q''\}$	l_{a10}: ldarg op1
$\{shift(Q'') \ \wedge \ s(0) = op1\}$	l_{a11}: ldarg op2
$\{shift^2(Q'') \ \wedge \ s(1) = op1 \ \wedge s(0) = op2\}$	l_{a12}: $binop_/$
$\{shift(Q'') \ \wedge \ s(0) = (op1/op2)\}$	l_{a13}: stloc Result
$\{Q'' \ \wedge \ Result = op1/op2\}$	l_{a14}: br l_{a20}
$\{op_2 \neq 0 \ \wedge \ operation \neq 1\}$	l_{a15}: call open_a_file
$\{op_2 \neq 0 \ \wedge \ operation \neq 1\}$	l_{a16}: ldarg op1
$\{op_2 \neq 0 \ \wedge \ operation \neq 1 \ \wedge s(0) = op1\}$	l_{a17}: ldarg op2
$\{op_2 \neq 0 \ \wedge \ operation \neq 1 \ \wedge \ s(1) = op1 \ \wedge \ s(0) = op2\}$	l_{a18}: $binop_*$
$\{op_2 \neq 0 \ \wedge \ operation \neq 1 \ \wedge \ s(0) = op1 * op2\}$	l_{a19}: stloc Result
$\{op_2 \neq 0 \ \wedge \ operation \neq 1 \ \wedge \ Result = op1 * op2\}$	l_{a20}: leave l_{c01}
	} // end .try
	.catch System.Exception {
$\left\{\left(\begin{array}{l} \left(\begin{array}{l} (operation = 1 \ \wedge \ op_2 \neq 0 \ \wedge \\ attempt \geq 0 \ \wedge \ attempt < 3) \end{array}\right) \vee \\ (operation \neq 1 \wedge \ op_2 \neq 0) \end{array}\right) \equiv R\right\}$	l_{b01}: ldarg operation
$\{shift(R) \ \wedge \ s(0) = operation\}$	l_{b02}: ldc 1
$\{shift^2(R) \ \wedge \ s(1) = operation \ \wedge \ s(0) = 1\}$	l_{b03}: $binop_=$
$\{shift(R) \ \wedge \ s(0) = (operation = 1)\}$	l_{b04}: brfalse l_{b10}
$\{(operation = 1 \wedge \ op_2 \neq 0 \ \wedge \ attempt \geq 0 \ \wedge \ attempt < 3) \equiv R''\}$	l_{b05}: ldloc attempt
$\{shift(R'') \ \wedge s(0) = attempt\}$	l_{b06}: ldc 1
$\{shift^2(R'') \ \wedge s(1) = attempt \ \wedge \ s(0) = 1\}$	l_{b07}: $binop_+$
$\{shift(R'') \ \wedge s(0) = attempt + 1\}$	l_{b08}: stloc attempt
$\{operation = 1 \wedge \ op_2 \neq 0 \ \wedge \ attempt \geq 0 \ \wedge \ attempt \leq 3\}$	l_{b09}: leave l_{a01}
$\{operation \neq 1 \wedge \ op_2 \neq 0\}$	l_{b10}: ldc -1
$\{operation \neq 1 \wedge \ op_2 \neq 0 \ \wedge \ s(0) = -1\}$	l_{b11}: stloc Result
$\{operation \neq 1 \wedge \ op_2 \neq 0 \ \wedge \ Result = -1\}$	l_{b13}: newobj instance void EIFFEL_EXCEPTION::.ctor (int32, string, class System.Exception)
$\left\{\begin{array}{l} operation \neq 1 \wedge \ op_2 \neq 0 \ \wedge \\ Result = -1 \ \wedge \ \tau(s(0)) \preceq Exception \end{array}\right\}$	l_{b14}: rethrow

Continued on next page

	} // end handler
$\left\{\begin{array}{l}(operation = 1 \land\ Result = op1//op2) \lor \\ (operation \neq 1 \land\ Result = op1 * op2)\end{array}\right\}$	l_{c01}: ldloc Result
$\left\{\left(\begin{array}{l}(operation = 1 \land\ Result = op1//op2) \lor \\ (operation \neq 1 \land\ Result = op1 * op2)\end{array}\right) s(0) = Result\right\}$	l_{c02}: ret

ensures

$$\left\{\begin{array}{l}\left(\begin{array}{l}(operation = 1 \land\ Result = op1//op2) \lor \\ (operation \neq 1 \land\ Result = op1 * op2)\end{array}\right) \lor \\ (excV \neq null\ \land\ operation \neq 1 \land\ op_2 \neq 0\ \land\ Result = -1)\end{array}\right\}$$

---

9 Soundness theorem

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

We prove the soundness of the proof translator and the specification translator. For the proof translator, it means that the translation produces valid bytecode proofs. However, it is not enough to prove that the translation produces a valid proof because the compiler could generate bytecode proofs where every precondition is false. The theorem expresses that

- if we have a valid source proof for the statement s_1, 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 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
- the retry postcondition implies the bytecode precondition at the label l_{retry}, and
- the exceptional postconditon in the source logic implies the bytecode precondition at the label l_{exc} but considering the value stored in the stack of the bytecode,

then we have to prove that every bytecode specification holds ($\vdash \{E_l\}\ I_l$).

The theorem is the following:

Theorem 1

$$\begin{array}{l}
\vdash \dfrac{Tree_1}{\{\ P\ \}\quad s_1\quad \{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \}} \quad \land \\
(I_{l_{start}}...I_{l_{end}}) = \nabla_S\left(\dfrac{Tree_1}{\{\ P\ \}\quad s_1\quad \{\ Q_n\ ,\ Q_r\ ,\ Q_e\ \}},\ l_{start}, l_{end+1}, l_{retry}, l_{exc}\right) \land \\
\left(Q_n \quad\Rightarrow\quad E_{l_{end+1}}\right) \quad \land \\
\left(Q_r \quad\Rightarrow\quad E_{l_{retry}}\right) \quad \land \\
(\ (Q_e\ \land\ excV \neq null\ \land\ s(0) = excV)\ \ \Rightarrow\ \ E_{l_{exc}})\ \ \land \\
\Rightarrow \\
\forall\ l\ \in\ l_{start}\ ...\ l_{end} : \vdash \{E_l\}\ I_l
\end{array}$$

The soundness proof of the specification translator has been formalized and proved in Isabelle. As well as the definition presented the the above Section, we have defined an evaluation function from Eiffel expressions to values. Given two heaps and a state, the theorem expresses if the expression e is well-formed then the value of the translation of the expression e is equal to the value returned by the evaluation of e. The theorem is the following:

Theorem 2

$$\begin{array}{l}
(\mathit{wellF}_b\ b) \Rightarrow (\mathit{value}_b\ b\ h_1\ h_2\ s) = ((\nabla_b\ b)\ h_1\ h_2\ s)\ \ \mathit{and} \\
(\mathit{wellF}_t\ t) \Rightarrow (\mathit{value}_t\ t\ h_1\ h_2\ s) = ((\nabla_t\ t)\ h_1\ h_2\ s)\ \ \mathit{and} \\
(\mathit{wellF}_{exp}\ e) \Rightarrow (\mathit{value}_{exp}\ e\ h_1\ h_2\ s) = ((\nabla_{exp}\ e)\ h_1\ h_2\ s)\ \ \mathit{and} \\
(\mathit{wellF}_{call}\ c) \Rightarrow (\mathit{value}_{call}\ c\ h_1\ h_2\ s) = ((\nabla_{call}\ c)\ h_1\ h_2\ s)\ \ \mathit{and} \\
(\mathit{wellF}_{arg}\ p) \Rightarrow (\mathit{value}_{arg}\ p\ h_1\ h_2\ s) = ((\nabla_{arg}\ p)\ h_1\ h_2\ s)
\end{array}$$

The proof of theorem 1 runs by induction on the structure of the derivation tree for $\{P\}\ s_1\ \{Q_n, Q_e\}$. The proof of theorem 2 is done in Isabelle. We present the proof of theorem 1 in appendix A.

10 Related Work

Necula and Lee [13] have developed certifying compilers, which produce proofs for basic safety properties such as type safety. The approach developed here supports interactive verification of source programs and as a result can handle more complex properties such as functional correctness.

Foundational Proof-Carrying Code has been extended by the open verifier framework for foundational verifiers [4]. It supports verification of untrusted code using custom verifiers. As in certifying compilers, the open verifier framework can prove basic safety properties.

Barthe *et al.* [3] show that proof obligations are preserved by compilation (for a non-optimizing compiler). They prove the equivalence between the verification condition (VC) generated over the source code and the bytecode. The translation in their case is less difficult because the source and the target languages are closer. This work does not address the translation of specifications.

Another development by the same group [2] translates 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, translate the source program into RTL; then, performed optimizations to build the appropriate certificate. This work involves a language that is simpler than ours and, like in the previously cited development, much closer to the target language than Eiffel is to CIL. We will investigate optimizing compilers as part of future work.

The Mobius project develops proof-transforming compilers [9]. They translate JML specifications and proof of Java source programs to Java Bytecode. The translation is simpler because the source and the target language are closer.

This work is based on our earlier effort [11] on proof-transforming compilation from Java to Bytecode. In that earlier project, the translation of method bodies is more complex due to the generated exception tables in Java bytecode. However, the source and the target langues are more similar than the languages used in this paper. Furthermore, our earlier work did not translate specifications.

11 Conclusion

We have defined proof transformation from a subset of Eiffel to bytecode. The PTC allows us to develop the proof in the source language (which is simpler), and transforms it into a bytecode proof. Due to Eiffel supports multiple inheritance and CIL does not, we focused on the translation of contracts. We showed that our translation is sound, that is, it produces valid bytecode proofs.

To show the feasibility of our approach, we implemented a PTC for a subset of Eiffel. The compiler takes a proof in XML format and produces the bytecode proof. The compiler is integrated to EiffelStudio.

As future work, we plan to develop a proof checker that tests the bytecode proof. Moreover, we plan to analyze how proofs can be translated using an optimizing compiler.

References

[1] F. Y. Bannwart and P. Müller. A Logic for Bytecode. In F. Spoto, editor, *Bytecode Semantics, Verification, Analysis and Transformation (BYTECODE)*, volume 141(1) of *ENTCS*, pages 255–273. Elsevier, 2005.

[2] G. Barthe, B. Grégoire, C. Kunz, and T. Rezk. Certificate Translation for Optimizing Compilers. In *13th International Static Analysis Symposium (SAS)*, LNCS, Seoul, Korea, August 2006. Springer-Verlag.

[3] G. Barthe, T. Rezk, and A. Saabas. Proof obligations preserving compilation. In *Third International Workshop on Formal Aspects in Security and Trust, Newcastle, UK*, pages 112–126, 2005.

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

[5] J. Gough. *Compiling for the .NET Common Language Runtime (CLR).* .NET series - Bertrand Meyer series editor. Prentice Hall, 2002.

[6] B. Meyer. Multi-language programming: how .net does it. In *3-part article in Software Development.* May, June and July 2002, especially Part 2, available at http://www.ddj.com/architect/184414864?

[7] B. Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

[8] B. Meyer (editor). ISO/ECMA Eiffel standard (Standard ECMA-367: Eiffel: Analysis, Design and Programming Language), June 2006. available at http://www.ecma-international.org/publications/standards/Ecma-367.htm.

[9] MOBIUS Consortium. Deliverable 4.3: Intermediate report on proof-transforming compiler. Available online from http://mobius.inria.fr, 2007.

[10] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS.* Springer-Verlag, 2002.

[11] P. Müller and M. Nordio. Proof-transforming compilation of programs with abrupt termination. In *SAVCBS '07: Proceedings of the 2007 conference on Specification and verification of component-based systems*, pages 39–46, 2007.

[12] G. Necula. *Compiling with Proofs.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1998.

[13] G. Necula and P. Lee. The Design and Implementation of a Certifying Compiler. In *Programming Language Design and Implementation (PLDI)*, pages 333–344. ACM Press, 1998.

[14] M. Pavlova. *Java Bytecode verification and its applications.* PhD thesis, University of Nice Sophia-Antipolis, 2007.

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

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

[17] A. Poetzsch-Heffter and N. Rauch. Soundness and Relative Completeness of a Programming Logic for a Sequential Java Subset. Technical report, Technische Universität Kaiserlautern, 2004.

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. Subsection A.2 presents the proof for the translation of compositional instruction. The proof for the translation of loop instructions is presented in subsection A.3. The proof for the translation of rescue and retry is presented in subsections A.5 and A.4 resp. The soundness proof for check and debug is presented in subsections A.6 and A.7 resp. Finally we present the proof for the translation of once functions in subsection A.8. We omit the proof of the translation of once routines because it is similar to the proof of once functions.

A.1 Notation

To make the proof easier to read, we write

$$\nabla_S\big(\{\ P\ \}\ \ s_1;s_2\ \ \{\ R_n\ ,\ R_r\ ,\ R_e\ \},\ l_{start}, l_{end}, l_{retry}, l_{exc}\big)$$

meaning:

$$\nabla_S\left(\frac{T_{S_1}\qquad\qquad T_{S_2}}{\left\{\ P\ \right\}\ \ s_1;s_2\ \ \left\{\ R_n\ ,\ R_r\ ,\ R_e\ \right\}},\ l_{start}, l_{end}, l_{retry}, l_{exc}\right) =$$

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

$$T_{S_1} \equiv \frac{T_1}{\left\{\ P\ \right\}\ \ s_1\ \ \left\{\ Q_n\ ,\ R_r\ ,\ R_e\ \right\}}$$

$$T_{S_2} \equiv \frac{T_2}{\left\{\ Q_n\ \right\}\ \ s_2\ \ \left\{\ R_n\ ,\ R_r\ ,\ R_e\ \right\}}$$

and we write

$$\vdash \{\ P\ \}\ \ s_1;s_2\ \ \{\ R_n\ ,\ R_r\ ,\ R_e\ \}$$

meaning

$$\left\{\ P\ \right\}\ \ s_1\ \ \left\{\ Q_n\ ,\ R_r\ ,\ R_e\ \right\}$$

$$\vdash \frac{\left\{\ Q_n\ \right\}\ \ s_2\ \ \left\{\ R_n\ ,\ R_r\ ,\ R_e\ \right\}}{\left\{\ P\ \right\}\ \ s_1;s_2\ \ \left\{\ R_n\ ,\ R_r\ ,\ R_e\ \right\}}$$

We use this notation in each proof of theorem 2.

A.2 Compositional instruction

The translation of compositional instruction was presented in section 6.5 on page 28.

We have to prove:

$$\begin{array}{l}
\vdash \dfrac{T_{S_1} \quad T_{S_2}}{\{\ P\ \}\quad s_1; s_2\ \ \{\ R_n\ ,\ R_r\ ,\ R_e\ \}} \quad \wedge \\
(I_{l_a} ... I_{l_b}) = \nabla_S\left(\dfrac{T_{S_1} \quad T_{S_2}}{\{\ P\ \}\quad s_1; s_2\ \ \{\ R_n\ ,\ R_r\ ,\ R_e\ \}}, l_{start}, l_{end+1}, l_{retry}, l_{exc}\right) \wedge \\
\big(R_n \quad \Rightarrow \quad E_{l_{b+1}}\big) \quad \wedge \\
\big(R_r \quad \Rightarrow \quad E_{l_{retry}}\big) \quad \wedge \\
(\ (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \quad \Rightarrow \quad E_{l_{exc}}) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_b : \vdash \{E_l\}\ I_l
\end{array}$$

By the first induction hypothesis we get:

$$\begin{array}{l}
\vdash \dfrac{Tree_1}{\{\ P\ \}\quad s_1\ \ \{\ Q_n\ ,\ R_r\ ,\ R_e\ \}} \quad \wedge \\
(I_{l_a} ... I_{l_{a\_end}}) = \nabla_S\left(\dfrac{tree_1}{\{\ P\ \}\quad s_1\ \ \{\ Q_n\ ,\ R_r\ ,\ R_e\ \}}, l_{start}, l_b, l_{retry}, l_{exc}\right) \wedge \\
\big(Q_n \quad \Rightarrow \quad E_{l_{a+1}}\big) \quad \wedge \\
\big(R_r \quad \Rightarrow \quad E_{l_{retry}}\big) \quad \wedge \\
(\ (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \quad \Rightarrow \quad E_{l_{exc}}) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_{a\_end} : \vdash \{E_l\}\ I_l
\end{array}$$

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

$$Q_n \quad \Rightarrow \quad E_{l_{a+1}} \tag{1}$$
$$R_r \quad \Rightarrow \quad E_{l_{retry}} \tag{2}$$
$$R_e \quad \Rightarrow \quad E_{l_{exc}} \tag{3}$$

(2) and (3) is proven by hypothesis. $E_{l_{a+1}}$ is equal to E_{l_b}. After the translation, the label precondition at l_b is Q_n. So we prove (1). Now we can apply the induction hypothesis and get:

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

Applying a similar reasoning to the second induction hypothesis we get:

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

Finally, we join both results and we get:

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

□

A.3 Loop instruction

The translation of loop instruction was presented in section 6.8.1 on page 32.

We have to prove:

$$\begin{array}{l}
\vdash \dfrac{T_{S_1} \quad T_{S_2}}{\{\ P\ \}\quad \textbf{from}\ s_1 ...\ \ \{\ I_n\ \wedge\ e\ ,\ Q_r\ ,\ R_e\ \}} \quad \wedge \\
(I_{l_a} ... I_{l_e}) = \nabla_S\left(\dfrac{T_{S_1} \quad T_{S_2}}{\{\ P\ \}\quad \textbf{from}\ s_1 ...\ \ \{\ I_n\ \wedge\ e\ ,\ Q_r\ ,\ R_e\ \}}, l_{start}, l_{end+1}, l_{retry}, l_{exc}\right) \wedge \\
\big((I_n\ \wedge\ e) \quad \Rightarrow \quad E_{l_{e+1}}\big) \quad \wedge \\
\big(Q_r \quad \Rightarrow \quad E_{l_{retry}}\big) \quad \wedge \\
(\ (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \quad \Rightarrow \quad E_{l_{exc}}) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_b : \vdash \{E_l\}\ I_l
\end{array}$$

By the first induction hypothesis we get:

$$\begin{array}{l}
\vdash \dfrac{Tree_1}{\{\ P\ \}\ \ s_1\ \ \{\ I_n\ ,\ Q_r\ ,\ R_e\ \}} \quad \wedge \\
(I_{l_a} ... I_{l_{a\_end}}) = \nabla_S \left(\dfrac{Tree_1}{\{\ P\ \}\ \ s_1\ \ \{\ I_n\ ,\ Q_r\ ,\ R_e\ \}},\ l_{start}, l_b, l_{retry}, l_{exc} \right) \ \wedge \\
(I_n \ \Rightarrow\ E_{l_b}) \quad \wedge \\
(Q_r \ \Rightarrow\ E_{l_{retry}}) \quad \wedge \\
(\ (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{l_{exc}}) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_{a\_end} : \vdash \{E_l\}\ I_l
\end{array}$$

E_{l_b} is I_n so $I_n \ \Rightarrow\ E_{l_b}$. $Q_r \ \Rightarrow\ E_{l_{retry}}$ and $(R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{l_{exc}}$ hold from the hypothesis. So we can apply the first induction hypothesis and get:

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

$\{I_n\}\ l_b$: br l_d holds due to the precondition of l_d is I_n. By the second induction hypothesis we get:

$$\begin{array}{l}
\vdash \dfrac{Tree_1}{\{\ \neg e\ \wedge\ I_n\ \}\ \ s_2\ \ \{\ I_n\ ,\ Q_r\ ,\ R_e\ \}} \quad \wedge \\
(I_{l_c} ... I_{l_{c\_end}}) = \nabla_S \left(\dfrac{T_{S_1}}{\{\ P\ \}\ \ s_1\ \ \{\ I_n\ ,\ Q_r\ ,\ R_e\ \}},\ l_c, l_d, l_{retry}, l_{exc} \right) \ \wedge \\
(I_n \ \Rightarrow\ E_{l_d}) \quad \wedge \\
(Q_r \ \Rightarrow\ E_{l_{retry}}) \quad \wedge \\
(\ (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{l_{exc}}) \\
\Rightarrow \\
\forall\ l\ \in\ l_c\ ...\ l_{c\_end} : \vdash \{E_l\}\ I_l
\end{array}$$

Due to E_{l_d} is equal to I_n and $Q_r \ \Rightarrow\ E_{l_{retry}}$ and $(R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{l_{exc}}$ hold from the hypothesis, we get:

$$\forall\ l\ \in\ l_c\ ...\ l_{c\_end} : \vdash \{E_l\}\ I_l$$

Applying a similar reasoning we get the proof for $nabla_E$. Finally, We have to proof $\{shift(I_n)\ \wedge\ s(0) = e\}\ l_e$: brfalse l_c. This hold because $l_c = \neg e\ \wedge\ I_n$. Then applying the definition of wp to brfalse we get $\{shift(I_n)\ \wedge\ s(0) = e\}$ implies $\{shift(I_n)\ \wedge\ s(0) = e\}$. Joining the proofs we get:

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

□

A.4 Retry instruction

The translation of retry instruction was presented in section 6.8.2 on page 32.

We have to prove:

$$\begin{array}{l}
\vdash \dfrac{}{\{\ P\ \}\ \ \textbf{retry}\ \ \{\ false\ ,\ P\ ,\ false\ \}} \quad \wedge \\
I_{l_a} = \nabla_S \left(\dfrac{}{\{\ P\ \}\ \ \textbf{retry}\ \ \{\ false\ ,\ P\ ,\ false\ \}},\ l_a, l_{a+1}, l_{retry}, l_{exc} \right) \ \wedge \\
(false \ \Rightarrow\ E_{l_{a+1}}) \quad \wedge \\
(P \ \Rightarrow\ E_{l_{retry}}) \quad \wedge \\
(\ (false\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{l_{exc}}) \\
\Rightarrow \\
\vdash \{P\}\ l_a :\ \text{br}\ l_{retry}
\end{array}$$

From the hypothesis we know $P \ \Rightarrow\ E_{l_{retry}}$. Then we can conclude $\vdash \{P\}\ l_a :$ br l_{retry}

□

A.5 Rescue

The translation of rescue was presented in section 6.8.3 on page 32.

We have to prove:

$$\begin{array}{l}
\vdash \dfrac{T_{S_1} \qquad T_{S_2}}{\{\ P\ \}\ \ s_1\ \textbf{rescue}\ s_2\ \ \{\ Q_n\ ,\ \mathit{false}\ ,\ R_e\ \}} \quad \wedge \\
(I_{l_a} ... I_{l_f}) = \nabla_S \left(\dfrac{T_{S_1} \qquad T_{S_2}}{\{\ P\ \}\ \ s_1\ \textbf{rescue}\ s_2\ \ \{\ Q_n\ ,\ \mathit{false}\ ,\ R_e\ \}},\ l_a, l_{f+1}, l_{retry}, l_{exc} \right) \ \wedge \\
(Q_n \ \Rightarrow \ E_{l_{f+1}}) \quad \wedge \\
(\mathit{false} \ \Rightarrow \ E_{l_{retry}}) \quad \wedge \\
(\ (R_e \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \ \Rightarrow \ E_{l_{exc}}) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_f : \vdash \{E_l\}\ I_l
\end{array}$$

By the first induction hypothesis we get:

$$\begin{array}{l}
\vdash \dfrac{Tree_1}{\{\ I_n\ \}\ \ s_1\ \ \{\ Q_n\ ,\ \mathit{false}\ ,\ I'_n\ \}} \quad \wedge \\
(I_{l_a} ... I_{l_{a\_end}}) = \nabla_S \left(\dfrac{Tree_1}{\{\ I_n\ \}\ \ s_1\ \ \{\ Q_n\ ,\ \mathit{false}\ ,\ I'_n\ \}},\ l_a, l_b, l_{retry}, l_c \right) \ \wedge \\
(Q_n \ \Rightarrow \ E_{l_b}) \quad \wedge \\
(\mathit{false} \ \Rightarrow \ E_{l_{retry}}) \quad \wedge \\
(\ (I'_n \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \ \Rightarrow \ E_{l_c}) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_{a\_end} : \vdash \{E_l\}\ I_l
\end{array}$$

Due to $E_{l_b} = Q_n$ and $E_{l_c} = (I'_n \ \wedge \ excV \neq null \ \wedge \ s(0) = excV)$ we can apply the induction hypothesis and get:

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

We can prove $\{Q_n\}\ l_b : \mathsf{leave}\ l_{next}$ and $\{I'_n \wedge excV \neq null \wedge s(0) = excV\}\ l_c\ \mathsf{stloc}\ \mathit{last\_exception}$ using the definition of wp^1_p presented in figure 3 on page 20.

By the second induction hypothesis we get:

$$\begin{array}{l}
\vdash \dfrac{Tree_2}{\{\ I'_n\ \}\ \ s_2\ \ \{\ R_e\ ,\ I_n\ ,\ R_e\ \}} \quad \wedge \\
(I_{l_d} ... I_{l_{d\_end}}) = \nabla_S \left(\dfrac{Tree_2}{\{\ I'_n\ \}\ \ s_2\ \ \{\ R_e\ ,\ I_n\ ,\ R_e\ \}},\ l_d, l_e, l_a, l_{exc} \right) \ \wedge \\
(R_e \ \Rightarrow \ E_{l_e}) \quad \wedge \\
(I_n \ \Rightarrow \ E_{l_a}) \quad \wedge \\
(\ (R_e \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \ \Rightarrow \ E_{l_{exc}}) \\
\Rightarrow \\
\forall\ l\ \in\ l_d\ ...\ l_{d\_end} : \vdash \{E_l\}\ I_l
\end{array}$$

To be able to apply the second induction hypothesis we have to prove:

$$\begin{array}{rcl}
R_e & \Rightarrow & E_{l_e} \\
I_n & \Rightarrow & E_{l_a} \\
(R_e \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) & \Rightarrow & E_{l_{exc}}
\end{array}$$

The first one holds because $E_{l_e} = R_e$. The second one holds due to $E_{l_a} = I_n$. And the third one holds from the hypothesis. So, we can apply the second induction hypothesis and get:

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

Finally, we have to prove $\{R_e\}\ l_e$: $\mathsf{ldloc}\mathit{last\_exception}$ and $\{R_e\ \wedge\ s(0) = \mathit{last\_exception}\}\ l_f$: rethrow. It can be proven using the definition of wp_p^1 presented in figure 3 on page 20.

Joining the proofs we get:

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

□

A.6 Check

The translation of once functions was presented in section 6.8.4 on page 33.

This proof is straight forward, we only have to use the definition of wp_p^1. We have to prove:

$$\begin{array}{l}
\vdash \overline{\{\ P\ \}\ \ \textbf{check}\ e\ \textbf{end}\ \ \{\ (P\ \wedge\ e\ )\ ,\ \mathit{false}\ ,\ (P\ \wedge\ \neg e\ )\ \}} \quad \wedge \\
(I_{l_a}...I_{l_d}) = \nabla_S\left(\overline{\{\ P\ \}\ \ \textbf{check}\ e\ \textbf{end}\ \ \{\ (P\ \wedge\ e\ )\ ,\ \mathit{false}\ ,\ (P\ \wedge\ \neg e\ )\ \}},\ l_a, l_{d+1}, l_{retry}, l_{exc} \right) \ \wedge \\
\big((P\ \wedge\ e\ )\ \Rightarrow\ E_{l_{d+1}}\big)\ \ \wedge \\
\big(\mathit{false}\ \ \Rightarrow\ \ E_{l_{retry}}\big)\ \ \wedge \\
(\ (P\ \wedge\ \neg e\ \wedge\ excV \neq \mathit{null}\ \wedge\ s(0) = excV)\ \ \Rightarrow\ \ E_{l_{exc}}) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_d : \vdash \{E_l\}\ I_l
\end{array}$$

To prove the instruction at l_b we have to show $\mathit{shisft}(P\ \wedge\ s(0) = e$ implies $wp_p^1(\mathsf{brtrue}l_{d+1})$. Applying the definition of wp_p^1, we have to prove:

$$(\mathit{shift}(P)\ \wedge\ s(0) = e)\ \ \Rightarrow\ \ (\neg s(0) \Rightarrow \mathit{shift}(E_c)) \wedge (s(0) \Rightarrow \mathit{shift}(E_{d+1}))$$

The first implication holds because $\mathit{shift}(E_c) = P\ \wedge\ \neg e$. The second implication is true due to from the hypothesis we know $(P\ \wedge\ e\ )\ \Rightarrow\ E_{l_{d+1}}$. Then using the definition of wp_p^1 we prove the implication.

The proof of instructions l_c and l_d is simple and only uses the definition of wp_p^1. Then, joining the proof we have:

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

□

A.7 Debug

The translation of once functions was presented in section 6.8.5 on page 33.

This proof is similar to the composition rule but we have two cases: $\mathcal{D} = \mathit{debug}$ and $\mathcal{D} = \mathit{not}_{\mathsf{debug}}$. In the first one, we use a similar reasoning that the compositional case and the knowledge that $\mathcal{D} = \mathit{debug}$. The second one is trivial because from the hypothesis we know P' implies E_{l_a}.

□

A.8 Once functions

The translation of once functions was presented in section 6.8.7 on page 35[2].

We have to prove:

[2] S, Q'_n and Q'_e are also defined on page 35

$$\begin{array}{l}
\vdash \dfrac{T_{body}}{\{\ S\ \}\ \ \{\mathbf{C}\}:\mathbf{f}(\mathbf{i})\ \ \{\ Q'_n\ ,\ \mathit{false}\ ,\ Q'_e\ \}} \quad \wedge \\
(I_{l_a} ... I_{l_n}) = \nabla_S \left(\dfrac{T_{body}}{\{\ S\ \}\ \ \{\mathbf{C}\}:\mathbf{f}(\mathbf{i})\ \ \{\ Q'_n\ ,\ \mathit{false}\ ,\ Q'_e\ \}},\ l_{start}, l_{end+1}, l_{retry}, l_{exc} \right) \ \wedge \\
\big(Q'_n \ \Rightarrow\ E_{l_{n+1}}\big) \quad \wedge \\
\big(\mathit{false} \ \Rightarrow\ E_{l_{retry}}\big) \quad \wedge \\
(\ (Q'_e \ \wedge\ excV \neq null \ \wedge\ s(0) = excV) \quad \Rightarrow \quad E_{l_{exc}}) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_n : \vdash \{E_l\}\ I_l
\end{array}$$

The proof for l_a is straightforward using the definition of wp_p^1. To prove l_b we have to prove the implication to l_c and l_i preconditions. The first one holds due to $\neg C\_f\_done$ holds then P holds. The second one is similar. l_c and l_d proof are also straightforward using the definition of wp_p^1.

Let T_{body} be

$$\dfrac{Tree_1}{\left\{ \begin{array}{l} P[\mathit{false}/C\_f\_done]\ \wedge \\ C\_f\_done \end{array} \right\}\ \ \{\mathbf{C}\}:\mathbf{f}(\mathbf{i})\ \ \{\ Q_n\ \wedge\ C\_f\_done\ ,\ \mathit{false}\ ,\ Q_e\ \wedge\ C\_f\_done\ \}}$$

By the first induction hypothesis we get:

$$\begin{array}{l}
\vdash T_{body} \quad \wedge \\
(I_{l_e} ... I_{l_{e\_end}}) = \nabla_S\ (T_{body},\ l_e, l_f, l_{retry}, l_g\ )\ \wedge \\
\big((Q_n\ \wedge\ C\_f\_done)\ \Rightarrow\ E_{l_f}\big) \quad \wedge \\
\big(\mathit{false} \ \Rightarrow\ E_{l_{retry}}\big) \quad \wedge \\
(\ (Q_e \ \wedge\ C\_f\_done\ \wedge\ excV \neq null \ \wedge\ s(0) = excV) \quad \Rightarrow \quad E_{l_g}) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_n : \vdash \{E_l\}\ I_l
\end{array}$$

Due to $E_{l_f} \equiv (Q_n \wedge C\_f\_done)$ and $E_{l_g} \equiv (\mathit{shift}(Q_e) \wedge C\_f\_done \wedge excV \neq null \wedge s(0) = excV)$ and $(\mathit{shift}(Q_e) \equiv Q_e$ because Q_e does not refer to the stack, we can apply the induction hypothesis and get:

$$\forall\ l\ \in\ l_e\ ...\ l_{e\_end} : \vdash \{E_l\}\ I_l$$

Finally, we have to prove $l_g, ...l_n$. The proof is straightforward using the definition of wp_p^1. Then we have prove:

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

□