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

Abstract. In modern development schemes the processing of programs often involves an intermediate step of translation to some intermediate bytecode, complicating the verification task. Expanding on the ideas of Proof-Carrying Code (PCC), we have built a proof-transforming compiler which translates a contract-equipped program and its proof into bytecode representing both the program and the proof; before execution starts, the program will be run through a proof checker. The proofs address not only security properties, as in the original PCC work, but full functional correctness as expressed by the original contracts. The task of the proof-transforming compiler is made particularly challenging by the impedance mismatch between the source language, Eiffel, and the target code, .NET CIL, which does not directly support such important Eiffel mechanisms as multiple inheritance and contract-based exceptions. We present the overall proof-transforming compilation architecture, the issues encountered, and the solutions that have been devised to bridge the impedance mismatch.

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

1 Introduction

The problem of software verification, hard enough in a traditional context, takes on new twists as advances in computing, designed to bring convenience and flexibility to users, also bring further headaches to verifiers. The work reported here addresses one such situation: verifying mobile code and other programs deployed through intermediate formats such as bytecode.

The problem arises because of the increased sophistication of our computing architectures. Along with new modes of computing arising from the role of the Internet, new modes of software deployment have emerged. Once we have written a program in a high-level language, instead of compiling it once and for all into machine code for execution on a given machine, we may generate intermediate code, often called "bytecode" (CIL on .NET, or JVM bytecode) and distribute it to numerous users who will execute it through either interpretation or a second phase of compilation known as "jitting". What then can and should we verify?

If we trust the interpreter or the jitter, the verification effort could apply to the bytecode; but this is a difficult proposition because typical bytecodes (CIL, JVM) discard some of the high-level information, in particular about types and control flow, that was present in the original and can be essential for a proof. In addition, proofs in the current state of the art can seldom be discharged in an entirely automatic fashion (for example by compilers, as a byproduct of the compilation process): they require interactive help from programmers. But then the target of the proof should be the program as written, not generated code which means nothing to the programmer. This suggests sticking to the traditional goal of proving correctness at the source level.

The problem now becomes to derive from a proof of the source code a guarantee of correctness of the generated bytecode. Unlike the interpreter or jitter, the compiler is often outside of the operating system; even if it is trusted, there is no guarantee against a third party tampering with the intermediate code. The notion of Proof-Carrying Code (PCC) [8] was developed to address this issue, with an original focus on safety properties: with PCC, a program producer develops code together with a formal proof (a certificate) that it possesses certain desirable properties. The program consumer checks the proof before executing the code or, in the above scheme, before interpreting or jitting it.

The original PCC work uses a certifying compiler [12] to prove simple safety properties automatically during the compilation process. The present work addresses the entire issue of functional correctness by introducing a proof-transforming compiler (PTC). The development scheme with this approach involves the following steps:

– Verify the source program, taking advantage of proof technology at the programming language level. This step can involve interaction with the programmer or verification expert.
– Translate both the source program and the proof into intermediate code, using the PTC. This step is automatic.
– Before a user runs the code (through interpretation or jitting), check the proof. This checking is again an automatic task; it can be performed by a simple proof-checking tool.

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 [5], 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 (interface only, implementation omitted) involving a reflective capability: the feature *type*, which gives the type of an object.

```
1 merge (other: LINKED_LIST [G]):LINKED_LIST [G]
          -- 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_INTERF* 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_INTERF* because in the target model decedents of the Eiffel class *LINKED_LIST* inherit from the interface *LINKED_LIST_INTERF* 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.

The main problems addressed in this paper are the definition of contract translation functions and proof translation functions from Eiffel to CIL. These translations are complex because CIL does not directly support important Eiffel mechanisms such as multiple inheritance and exceptions with rescue blocks. To be able to translate both contracts and proofs, we use deeply-embedded Eiffel expressions in the contracts and the source proof. The main contributions of this paper are: (1) a contract translation function from Eiffel to CIL which handles the lack of multiple inheritance in CIL; (2) a proof translation function from Eiffel to CIL which maps rescue blocks into CIL instructions.

The rest of this paper explores the translation of programs and associated proofs: issues such as the above example, and the solutions that we have adopted.

Section 2 surveys the semantics of the source language, Eiffel, by presenting a Hoare-style logic; section 3 does the same for the target language, CIL. Section 4 presents the specification translator. Section 5 defines the proof transformation process. Section 6 illustrates this transformation through an example. Section 7 introduces a soundness theorem. Section 8 discusses related work, and Section 9 summarizes the result and describes future developments.

2 Source language and logic

In this section, we present the Eiffel subset used in this paper and summarize the logic that is used for the verification of Eiffel programs.

2.1 Basics

The source language is a subset of Eiffel [8] with the following syntax:

```
exp     ::= literal | var | exp op exp
instr   ::= x := exp | instr; instr
          | from instr until exp loop instr end
          | if exp then instr else instr end
routine ::= name (var : Type) : Type is
              require boolExp
              [ local var : Type, ... ]
              do
                instr
              [ rescue
                instr ]
              ensure boolExp
            end
```

Once routines are not included. Exceptions are, but expression evaluation cannot cause an exception.

Since exceptions raise some of the most interesting translation problems, the following reminder of Eiffel exception semantics is in order. The ideas behind exception handling in Eiffel (see [6]) are based on Design by Contract principles. A routine execution either succeeds - meaning it achieves its contract - or fails, triggering an exception. An exception is, more generally, an abnormal event during that execution, due for example in turn to the failure of a routine that it has called. The exception causes execution of the routine's `rescue` clause (either explicit or default). If at the end of the clause the variable `Retry` has value true, the normal routine body (do clause) is executed again, in a new attempt to satisfy the contract. If not, the routine execution failed, triggering an exception that can be handled through the `rescue` clause of the caller. This scheme implies a clear separation of roles between the do and `rescue` clauses: only the former is charged with achieving the routine's contract, as stated by the postcondition. The `rescue` clause only concerns itself with trying to correct the situation that led to the exception; in addition, it must, if it cannot `Retry`, re-establish the class invariant so as to leave the object in a consistent state for future calls.

Note that this specification slightly departs from the current Eiffel standard, where `Retry` is an instruction, not a variable. The change was suggested by our semantic work [7] and will be adopted by a future revision of the language standard. Assignments to `Retry` can appear in either a do clause or a `rescue` clause; if its value is true at the end of exception processing the routine re-executes its body, otherwise it fails, triggering a new exception.

2.2 Routine and instruction specifications

This paper focuses on the aspects of the translation framework that are most interesting for the translation of proofs. A technical report [13] gives the details for such other object-oriented mechanisms as object creation, attribute access, and routine call.

The logic for the source language is based on the programming logic introduced in [10, 16], adapted for Eiffel and extended with new rules for the `rescue`/`Retry` exception mechanism.

Poetzsch-Heffter et al. [17] use a special variable χ to capture the status of the program, with values such as normal and exceptional. This variable is not necessary in the bytecode proof since non-linear control flow is implemented via jumps. To eliminate the variable, we use Hoare triples with two postconditions: one for normal termination, the other for exceptions. This simplifies both the translation and the presentation.

The specification of a routine, or more generally an instruction S, is a Hoare triple of the form $\{ P \} \;\; S \;\; \{ Q_n \; , \; Q_e \}$, where $P, \; Q_n, \; Q_e$ are deeply-embedded Eiffel expressions extended with universal and existential quantifiers, and S is a routine or an instruction. 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* or *instruction specification* depending on whether S is a routine or instruction.

To make proof translation feasible in the presence of changes to the type structure, it is essential that both preconditions and postconditions of the Hoare triples be deeply-embedded Eiffel expressions. A deep embedding preserves the syntactic structure of the expression, which we exploit during the translation of Eiffel types to CIL types.

A specification $\{ P \} \;\; S \;\; \{ Q_n \; , \; Q_e \}$ defines the following refined partial correctness property [15]: if S's execution starts in a state satisfying P, then one of the following holds: (1) S terminates normally in a state where Q_n holds, or S triggers an exception and Q_e holds, or (2) S aborts due to errors or actions that are beyond the semantics of the programming language, for instance, memory allocation problems, or (3) S runs forever.

2.3 Axiomatic semantics

The axiomatic semantics consists of the axioms and rules for instructions and routines, as well as several language-independent rules such as the rule of consequence, allowing the strengthening of preconditions and weakening of postconditions. Figure 1 shows the rules for compound instructions, loops and `rescue` clauses. The compound and loop rules are standard. The rescue rule is one of the contributions of this paper.

In a compound, s_1 executes first; then s_2 executes if and only if s1 has terminated normally. In a loop, s_1 executes. If s_1 causes an exception then the postcondition of the loop is the postcondition of s_1 (R_e). If s_1 terminates normally and the condition e does not hold, then the body of the loop (s_2) executes. If s_2 terminates normally then the invariant I holds. If s_2 triggers an exception, R_e holds.

The `rescue` rule applies to any routine with a `rescue` clause. The following informal reminder of the Eiffel exception mechanism: if s_1 terminates normally then the `rescue` block is not executed and the postcondition is Q_n . If s_1 triggers an exception, the `rescue` block executes. If the instruction s_2 terminates normally and the `Retry` variable is true then control flow transfers back to the beginning of the routine and I_r holds. If s_2 terminates normally and `Retry` is false, the routine triggers the "routine failure" exception and R_e holds. If both s_1 and s_2 trigger an exception, the last one takes precedence, and R_e holds.

This rule interprets a `rescue` clause as a loop that iterates from s_1 loop $s_2; s_1$ until s_1 causes no exception or `Retry` is set to false. Note that the loop body is executed only if s_1 triggers an exception. The invariant I_r is the loop invariant, called retry invariant in this context.

Compound

$$\frac{\begin{array}{c}\left\{P\right\}\ s_1\ \left\{Q_n\ ,\ R_e\right\}\\ \left\{Q_n\right\}\ s_2\ \left\{R_n\ ,\ R_e\right\}\end{array}}{\left\{P\right\}\ s_1; s_2\ \left\{R_n\ ,\ R_e\right\}}$$

Rescue clause

$$\frac{\begin{array}{c}P \Rightarrow I_r\\ \left\{I_r\right\}\ s_1\ \left\{Q_n\ ,\ Q_e\right\}\\ \left\{Q_e\right\}\ s_2\ \left\{\mathit{Retry} \Rightarrow I_r\ \wedge\ \neg \mathit{Retry} \Rightarrow R_e\ ,\ R_e\right\}\end{array}}{\left\{P\right\}\ \texttt{do}\ s_1\ \texttt{rescue}\ s_2\ \left\{Q_n\ ,\ R_e\right\}}$$

Loop

$$\frac{\begin{array}{c}\left\{P\right\}\ s_1\ \left\{I\ ,\ R_e\right\}\\ \left\{\neg e\ \wedge\ I\right\}\ s_2\ \left\{I\ ,\ R_e\right\}\end{array}}{\left\{P\right\}\ \texttt{from}\ s_1\ \texttt{until}\ e\ \texttt{loop}\ s_2\ \texttt{end}\ \left\{(I\ \wedge\ e)\ ,\ R_e\right\}}$$

Fig. 1. Rules for compound, rescue clause, and loop.

3 Bytecode language and logic

The bytecode language consists of interfaces and classes. Each class consists of methods and fields. Methods are a sequence of labeled bytecode instructions, which operate on the operand stack, local variables, arguments, and the heap.

3.1 Bytecode basics

The bytecode language we use is a slight variant of CIL. We treat local variables and routine arguments using the same instructions. Instead of using an array of local variables like in CIL, we use the name of the source variable. Furthermore, to simplify the translation, we assume the bytecode language has a type boolean. The bytecode instructions and their informal description are the following:

– ldc v: pushes constant v onto the stack
– ldloc x: pushes the value of a variable x onto the stack
– stloc x: pops the topmost element off the stack and assigns it to the local variable x
– bin_{op}: removes the two topmost values from the stack and pushes the result of applying bin_{op} to these values
– br l: transfers control to the point l
– brfalse l: transfers control to the point l if the topmost element of the stack is false and unconditionally pops it
– rethrow: takes the topmost value from the stack, assumed to be an exception, and rethrows it
– leave l: exit from the try or catch block to the point l

3.2 Method and instruction specifications

The bytecode logic we use is the logic developed by Bannwart and Müller [1]. It is a Hoare-style program logic, which is similar in its structure to the source logic. In particular, both logics treat methods in the same way, contain the same language-independent rules, and triples have a similar meaning. These similarities make proof transformation feasible.

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

Each instruction is treated individually in the logic since the unstructured control flow of bytecode programs makes it difficult to handle instruction sequences. Each individual instruction I_l in a method body p has a precondition E_l. An instruction with its precondition is called an *instruction specification*, written as $\{E_l\}\ l : I_l$.

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

3.3 Rules

Assertions refer to the current stack, arguments, local variables, and the heap. The current stack is referred to as s and its elements are denoted by non-negative integers: element 0 is the topmost element, etc. The interpretation $[E_l] : \mathit{State} \times \mathit{Stack} \rightarrow \mathit{Value}$ for s is defined as follows: $[s(0)]\langle S, (\sigma, v)\rangle = v$ and $[s(i+1)]\langle S, (\sigma, v)\rangle = [s(i)]\langle S, \sigma\rangle$.

The functions *shift* and *unshift* define the substitutions that occur when values are pushed onto and popped from the stack, respectively. Their definitions are the following: $\mathit{shift}(E) = E[s(i+1)/s(i)$ for all $i \in \mathbb{N}]$ and $\mathit{unshift} = \mathit{shift}^{-1}$.

The rules for instructions have the following form:

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

where $wp(I_l)$ denotes the *local weakest precondition* of instruction I_l. The rule specifies that E_l (the precondition of I_l) has to imply the weakest precondition of I_l with respect to all possible successor instructions of I_l. The precondition E_l denotes the precondition of the instruction I_l. The precondition E_{l+1} denotes the precondition of I_l's successor instruction. Table 1 shows the definition of wp.

I_l	$wp(I_l)$
ldc v	$\mathit{unshift}(E_{l+1}[v/s(0)])$
ldloc x	$\mathit{unshift}(E_{l+1}[x/s(0)])$
stloc x	$(\mathit{shift}(E_{l+1}))[s(0)/x]$
bin_{op}	$(\mathit{shift}(E_{l+1}))[s(1)\ op\ s(0)/s(1)]$
br l'	$E_{l'}$
brfalse l'	$(s(0) \Rightarrow \mathit{shift}(E_{l+1})) \wedge (\neg s(0) \Rightarrow \mathit{shift}(E_{l'}))$
leave l'	$E_{l'}$

Table 1. Definition of function *wp*.

4 Specification translator

The specification translator translates Eiffel contracts into first order logic (FOL). The challenging problem in the specification translator is produced by the impedance mismatch between Eiffel and CIL. Due to CIL does not directly support multiple inheritance, Eiffel classes are mapped to interfaces or implementation classes. To be able to translate contracts, we use deeply-embedded Eiffel expressions.

4.1 Translation basics

In Hoare triples, pre- and postconditions may refer to the structure of the Eiffel program. Therefore, in our logic, pre- and postconditions are deeply-embedded Eiffel expressions, extended with universal and existential quantifiers. The proof translation proceeds in two steps: first, translation of pre-and postconditions into FOL using the translation function presented in this section; then, translation of the proof using the functions presented in Section 5.

We have defined a deep embedding of the Eiffel expressions used in the contract language. Then, we have defined translation functions to FOL. The datatype definitions, the translation functions and their soundness proof are formalized in Isabelle. In this section, we present the most interesting definitions and formalizations, for a complete definition see [13].

4.2 Datatype definitions

Eiffel contracts are based on boolean expressions, extended (for postconditions) with the old notation. They can be constructed using the logical operators $\neg$ and $\vee$, equality, and the type functions *ConformsTo* or *IsEqual*. Expressions are constants, local variables and arguments, attributes, routine calls, creation expressions, old expressions, boolean expressions, and *Void*. Arguments are treated as local variables using the sort *RefVar* to minimize the datatype definition. Furthermore, boolean variables are not introduced in the definition *boolExp*. They are treated as local variables using the sort *RefVar*. We assume routines have exactly one argument.

```
datatype EiffelContract = Require  boolExpr
                        | Ensure  boolExpr
datatype boolExpr = Const  bool
                  | Neg  boolExpr
                  | Or  boolExpr boolExpr
                  | Eq    expr expr
                  | Type  typeFunc

datatype typeFunc = ConformsTo  typeExpr typeExpr
                  |  IsEqual  typeExpr typeExpr
datatype typeExpr = EType  EiffelType
                  | Type  expr
datatype expr = ConstInt  int
              | RefVar  var
              | Att  objID attrib
              | 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 class identifier, 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, attributes and routines are defined as follows:

```
datatype var     = Var  vID EiffelType
                 | Result  EiffelType
                 | Current  EiffelType
datatype attrib  = Attr  (aID : attribID) EiffelType
datatype routine = Routine  routineID EiffelType EiffelType
```

4.3 Object store and values

An object store is modeled by an abstract data type *store*. We use the object store presented by Poetzsch-Heffter [15]. The Eiffel object store and the CIL object store are the same. The following operations apply to the object store: $accessC(os, l)$ 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:

$$\begin{array}{ll} accessC :: & store \rightarrow location \rightarrow value \\ alive :: & value \rightarrow store \rightarrow bool \\ alloc :: & store \rightarrow classID \rightarrow store \\ new :: & store \rightarrow classID \rightarrow value \\ update :: & store \rightarrow location \rightarrow value \rightarrow store \end{array}$$

The axiomatization of these functions is presented by Poetzsch-Heffter [15].

A value is a boolean, an integer, the void value, or an object reference. An object is characterized by its class and an identifier of infinite sort *objID*.

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

4.4 Mapping Eiffel types to CIL

To define the translation from Eiffel contracts to *FOL*, it is useful first to define CIL types and mapping functions that map Eiffel types to the CIL types: boolean, integer, interfaces, classes and the null type.

```
datatype CilType = CilBoolean
                 | CilInteger
                 | Interface classID
                 | CilClass classID
                 | NullT
```

The translation then uses two functions that map Eiffel types to CIL: (1) $\nabla_{interface}$ maps an Eiffel type to a CIL interface; (2) ∇_{class} maps the type to a CIL implementation class. These functions are defined as follows:

$$
\begin{array}{llll}
\multicolumn{2}{l}{\nabla_{interface} :: \ \mathit{EiffelType} \rightarrow \ \mathit{CilType}} & \multicolumn{2}{l}{\nabla_{class} :: \ \mathit{EiffelType} \rightarrow \ \mathit{CilType}} \\
\quad \nabla_{interface}(\mathbf{Boolean}) & = \mathbf{CilBoolean} & \quad \nabla_{class}(\mathbf{Boolean}) & = \mathbf{CilBoolean} \\
\quad \nabla_{interface}(\mathbf{Integer}) & = \mathbf{CilInteger} & \quad \nabla_{class}(\mathbf{Integer}) & = \mathbf{CilInteger} \\
\quad \nabla_{interface}(\mathbf{EClass}\ n) & = \mathbf{Interface}\ n & \quad \nabla_{class}(\mathbf{EClass}\ n) & = \mathbf{CilClass}\ n \\
\quad \nabla_{interface}(\mathbf{None}) & = \mathbf{NullT} & \quad \nabla_{class}(\mathbf{None}) & = \mathbf{NullT}
\end{array}
$$

The translation of routine calls needs method signatures in CIL and a translation function that maps Eiffel routines to CIL methods. The function $\nabla_{interface}$ serves to map types t_1 and t_2 to CIL types.

$\mathbf{datatype}\ \mathit{CilMethod} = \mathbf{Method}\ \mathit{methodID}\ \mathit{CilType}\ \mathit{CilType}$

$\nabla_r ::\ \mathit{routine} \rightarrow\ \mathit{CilMethod}$

$\quad \nabla_r(\mathbf{Routine}\ n\ t1\ t2) = (\mathbf{Method}\ \ n\ \ (\nabla_{interface}\ t1)\ \ (\nabla_{interface}\ t2))$

4.5 Contract translation

The translation of the specification relies on 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. These functions use two object stores, the second one is used to evaluate old expressions. *state* is a mapping from variables to values ($var \rightarrow value$). The signatures of these functions are the following:

$$
\begin{array}{l}
\nabla_b :: \mathit{boolExpr} \rightarrow (\mathit{store} \rightarrow \mathit{store} \rightarrow \mathit{state} \rightarrow \mathit{value}) \\
\nabla_{exp} :: \mathit{expr} \rightarrow (\mathit{store} \rightarrow \mathit{store} \rightarrow \mathit{state} \rightarrow \mathit{value}) \\
\nabla_t :: \mathit{typeFunc} \rightarrow (\mathit{store} \rightarrow \mathit{store} \rightarrow \mathit{state} \rightarrow \mathit{value}) \\
\nabla_{call} :: \mathit{callRoutine} \rightarrow (\mathit{store} \rightarrow \mathit{store} \rightarrow \mathit{state} \rightarrow \mathit{value}) \\
\nabla_{arg} :: \mathit{argument} \rightarrow (\mathit{store} \rightarrow \mathit{store} \rightarrow \mathit{state} \rightarrow \mathit{value})
\end{array}
$$

The definition of the function ∇_b is the following:

$$
\begin{array}{ll}
\nabla_b(\mathbf{Const}\ b) & = \lambda\ (h_1, h_2 :: \mathit{store})\ (s :: \mathit{state}) :\ (\mathit{BoolV}\ \ b) \\
\nabla_b(\mathbf{Neg}\ b) & = \lambda\ (h_1, h_2 :: \mathit{store})\ (s :: \mathit{state}) : \\
 & \quad (\mathit{BoolV}\ \neg(aB(\nabla_b\ b\ h_1\ h_2\ s))) \\
\nabla_b(\mathbf{Or}\ b_1\ b_2) & = \lambda\ (h_1, h_2 :: \mathit{store})\ (s :: \mathit{state}) : \\
 & \quad (\mathit{BoolV}\ \ (aB(\nabla_b\ b_1\ h_1\ h_2\ s)) \vee (aB(\nabla_b\ b_2\ h_1\ h_2\ s))) \\
\nabla_b(\mathbf{Eq}\ e_1\ e_2) & = \lambda\ (h_1, h_2 :: \mathit{store})\ (s :: \mathit{state}) : \\
 & \quad (\mathit{BoolV}\ \ (aI(\nabla_{exp}\ e_1\ h_1\ h_2\ s)) = (aI(\nabla_{exp}\ e_2\ h_1\ h_2\ s))) \\
\nabla_b(\mathbf{Type}\ e) & = \lambda\ (h_1, h_2 :: \mathit{store})\ (s :: \mathit{state}) :\ (\nabla_t\ e\ h_1\ h_2\ s))
\end{array}
$$

The function ∇_t maps the Eiffel types to CIL. The Eiffel function *ConformsTo* is mapped to the function $\preceq_c$ (subtyping in CIL). Its types are translated to interfaces using the function $\nabla_{interface}$. The function *IsEqual* is translated using the function $=$ (types equality in CIL). Its types are translated to CIL classes using the function ∇_{class}. The function ∇_t is defined as follows:

$\nabla_t(\mathbf{ConformsTo}\ t_1\ t_2) = \lambda\ (h_1, h_2 :: \mathit{store})(s :: \mathit{state}) :$
$\qquad (\mathit{BoolV}(\nabla_{\mathit{interface}}(\nabla_{\mathit{type}} t_1)) \preceq_c (\nabla_{\mathit{interface}}(\nabla_{\mathit{type}} t_2)))$
$\nabla_t(\mathbf{IsEqual}\ t_1\ t_2) = \lambda\ (h_1, h_2 :: \mathit{store})(s :: \mathit{state}) :$
$\qquad (\mathit{BoolV}(\nabla_{\mathit{class}}(\nabla_{\mathit{type}} t_1)) = (\nabla_{\mathit{class}}(\nabla_{\mathit{type}} t_2)))$

The function ∇_{type} given a type expression returns its Eiffel type:
$\nabla_{type} ::\ \mathit{typeExp} \rightarrow\ \mathit{EiffelType}$
$\quad \nabla_{type}(\mathbf{EType}\ t) = t$
$\quad \nabla_{type}(\mathbf{Expression}\ e) = (\mathit{typeOf}\ \ e)$

The function ∇_{exp} translates local variables using the *state* s. Creation instructions are translated using the functions *new* and *alloc*. The translation of old expressions uses the second *store* to map the expression e to CIL. The definition is:
$\nabla_{exp}(\mathbf{ConstInt}\ i) = \lambda\ (h_1, h_2 :: \mathit{store})(s :: \mathit{state}) : (\mathit{IntV}\ \ i)$
$\nabla_{exp}(\mathbf{RefVar}\ v) = \lambda\ (h_1, h_2 :: \mathit{store})(s :: \mathit{state}) : (s(v))$
$\nabla_{exp}(\mathbf{Att}\ ob\ a) = \lambda\ (h_1, h_2 :: \mathit{store})(s :: \mathit{state}) :$
$\qquad (\mathit{accessC}\ h_1\ (\mathit{Loc}\ (\mathit{aID}\ a)\ ob))$
$\nabla_{exp}(\mathbf{CallR}\ crt) = \lambda\ (h_1, h_2 :: \mathit{store})(s :: \mathit{state}) :$
$\qquad (\nabla_{call}\ crt\ h_1\ h2\ s)$
$\nabla_{exp}(\mathbf{Create}\ t\ rt\ p) = \lambda\ (h_1, h_2 :: \mathit{store})(s :: \mathit{state}) :$
$\qquad (\mathit{new}\ (\mathit{alloc}\ h_1\ (\mathit{cID}\ t))\ (\mathit{cID}\ t))$
$\nabla_{exp}(\mathbf{Old}\ e) = \lambda\ (h_1, h_2 :: \mathit{store})(s :: \mathit{state}) :$
$\qquad (\nabla_{exp}\ e\ h_2\ h_2\ s)$
$\nabla_{exp}(\mathbf{Bool}\ b) = \lambda\ (h_1, h_2 :: \mathit{store})(s :: \mathit{state}) :$
$\qquad (\nabla_b\ b\ h_1\ h_2\ s)$
$\nabla_{exp}(\mathbf{Void}) = \lambda\ (h_1, h_2 :: \mathit{store})(s :: \mathit{state}) : (\mathit{VoidV})$

The function ∇_{call} is defined as follows:
$\nabla_{call}(\mathbf{Call}\ e_1\ rt\ p) = \lambda\ (h_1, h_2 :: \mathit{store})(s :: \mathit{state}) :$
$\qquad (\mathbf{CilInvokeVal}\ h_1\ (\nabla_r\ rt)\ (\nabla_{exp}\ e_1\ h_1\ h_2\ s)(\nabla_{arg}\ p\ h_1\ h_2\ s))$

The function *CilInvokeVal* takes a CIL method m and two values (its argument p and invoker e_1) and returns the value of the result of invoking the method m with the invoker e_1 and argument p.
The definition of the function ∇_{arg} is the following:
$\nabla_{arg}(\mathbf{Argument}\ e) = \lambda\ (h_1, h_2 :: \mathit{store})(s :: \mathit{state}) :\ (\nabla_{exp}\ e\ h_1\ h_2\ s)$

4.6 Example translation

To be able to translate contracts, first we embed the contracts in Isabelle using the above data type definitions. Then, we apply the translation function ∇_b which produces the contracts in FOL. Following, we present the embedding of the contracts of the function *merge* presented in Section 1. Its precondition is embedded as follows:

Type (**ConformsTo** (Type (**RefVar** *other*)) (EType *LINKED_LIST*[*G*]))
Type (**IsEqual** (Type (**RefVar** *Current*)) (Type (**RefVar** *other*)))

The deep embedding of *merge*'s postcondition is as follows:

$$\text{Type} \; (\; \textbf{IsEqual} \; (\text{Type} \; (\textbf{RefVar} \; \textit{Current}) \;) \; (\text{EType} \; \textit{LINKED\_LIST}[G]) \;)$$

The application of the function ∇_b to the precondition produces the following expression:

$$\begin{array}{l}\lambda \; (h_1, h_2 :: \textit{store})(s :: \textit{state}) : \\ \quad \textit{BoolV}(\textit{typeOf} \;\; \textit{other}) \; \preceq_c (\textbf{interface} \; \textit{LINKED\_LIST}[G]) \\ \lambda \; (h_1, h_2 :: \textit{store})(s :: \textit{state}) : \;\; \textit{BoolV}(\textit{typeOf} \;\; \textit{Current}) \; = (\textit{typeOf} \;\; \textit{other})\end{array}$$

The result of the application of the function ∇_b to the deep embedding of *merge*'s postcondition is the following:

$$\begin{array}{l}\lambda \; (h_1, h_2 :: \textit{store})(s :: \textit{state}) : \\ \quad \textit{BoolV}(\textit{typeOf} \;\; \textit{Current}) \; = (\textbf{CilClass} \; \textit{LINKED\_LIST}[G])\end{array}$$

In the precondition, the type *LINKED_LIST[G]* is translated to the interface *LINKED_LIST[G]* because the precondition uses the function *ConformsTo*. However, in the postcondition, the type *LINKED_LIST[G]* is translated to the class *LINKED_LIST[G]* because it uses the function *IsEqual*. The PTC can translates these types because it takes deeply-embedded Eiffel expressions as input.

5 Proof translation

Our proof translator is based on two transformation functions, ∇_S and ∇_E, for instructions and expressions, respectively. Each yields a sequence of bytecode instructions and their specifications.

5.1 Transformation function basics

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

$$\begin{array}{l}\nabla_E \; : \textit{Precondition} \times \textit{Expression} \times \textit{Postcondition} \times \textit{Label} \rightarrow \textit{Bytecode\_Proof} \\ \nabla_S \; : \textit{Proof\_Tree} \times \textit{Label} \times \textit{Label} \times \textit{Label} \rightarrow \textit{Bytecode\_Proof}\end{array}$$

In ∇_E the label is used as the starting label of the translation. Proof_Tree is a derivation in the source logic. In ∇_S, the three labels are: (1) *start* for the first label of the resulting bytecode; (2) *next* for the label after the resulting bytecode; this is for instance used in the translation of an `else` branch to determine where to jump at the end; (3) *exc* for the jump target when an exception is thrown. The *Bytecode_Proof* type is defined as a list of instruction specifications.

The proof translation will now be presented for the compound instruction, loops, and `rescue` clauses. The definition of ∇_E is simple, it translates expressions to CIL proofs. Due to space limitations, this definition is not presented here (see our technical report [13]). Furthermore, in this technical report, the translation also includes object-oriented features such as object creation and routine invocation.

5.2 Compound Instruction

Compound 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:

$$T_{S_1} \equiv \frac{Tree_1}{\{\,P\,\} \quad s_1 \quad \{\,Q_n\ ,\ R_e\,\}} \qquad T_{S_2} \equiv \frac{Tree_2}{\{\,Q_n\,\} \quad s_2 \quad \{\,R_n\ ,\ R_e\,\}}$$

The definition of the translation is the following:

$$\begin{array}{c}\nabla_S \left(\dfrac{T_{S_1} \qquad\qquad T_{S_2}}{\{\,P\,\} \quad s_1; s_2 \quad \{\,R_n\ ,\ R_e\,\}},\ l_{start}, l_{next}, l_{exc} \right) = \\ \nabla_S\,(T_{S_1},\ l_{start}, l_b, l_{exc}) \\ \nabla_S\,(T_{S_2},\ l_b, l_{next}, l_{exc})\end{array}$$

The bytecode for s_1 establishes Q_n, which is the precondition of the first instruction of the bytecode for s_2. Therefore, the concatenation of the bytecode as the result of the translation of s_1 and s_2, produces a sequence of valid instruction specifications. Section 7 will discuss soundness.

5.3 Loop Instruction

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

$$T_{S_1} \equiv \frac{Tree_1}{\{\,P\,\} \quad s_1 \quad \{\,I\ ,\ R_e\,\}} \qquad T_{S_2} \equiv \frac{Tree_2}{\{\,\neg e\ \wedge\ I\,\} \quad s_2 \quad \{\,I\ ,\ R_e\,\}}$$

The first step of translating the loop is to translate s_1 using ∇_S. Then, control is transferred to l_d where the loop expression is evaluated. The body of the loop is translated with l_c. The loop invariant holds at the begging of the loop expression evaluation (at l_d). The definition is:

$$\nabla_S \left(\dfrac{T_{S_1} \qquad\qquad T_{S_2}}{\{\,P\,\} \quad \begin{array}{l}\texttt{from } s_1 \texttt{ until } e \\ \texttt{loop } s_2 \texttt{ end}\end{array} \quad \{\,(I\ \wedge\ e)\ ,\ R_e\,\}},\ l_{start}, l_{next}, l_{exc} \right) =$$

$$\begin{array}{ll} & \nabla_S\,(\ T_{S_1},\ l_{start}, l_b, l_{exc}\ ) \\ \{I\} & l_b : \mathsf{br}\ l_d \\ & \nabla_S\,(\ T_{S_2},\ l_c, l_d, l_{exc}\ ) \\ & \nabla_E(\ I,\ e,\ \{shift(I)\ \wedge\ s(0) = e\},\ l_d\ ) \\ \{shift(I)\ \wedge\ s(0) = e\} & l_e : \mathsf{brfalse}\ l_c \end{array}$$

5.4 Rescue clause

The translation of rescue clauses to CIL is one of the most interesting translations. Since rescue clauses do not exist in CIL, this translation maps rescue clauses to .try and catch CIL instructions. Let T_{S_1} and T_{S_2} be the following proof trees:

$$T_{S_1} \equiv \frac{\mathit{Tree}_1}{\{\, I_r \,\}\quad s_1 \quad \{\, Q_n \;,\; Q_e \,\}} \qquad T_{S_2} \equiv \frac{\mathit{Tree}_2}{\{\, Q_e \,\}\quad s_2 \quad \left\{ \begin{array}{l} \mathit{Retry} \Rightarrow I_r \,\wedge \\ \neg \mathit{Retry} \Rightarrow R_e \end{array} \;,\; R_e \right\}}$$

First, the instruction s_1 is translated to 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 into a catch block. For this, the exception object is first stored in a temporary variable and then s_2 is translated. In this translation, the `Retry` label is updated to l_{start} (the beginning of the routine). Finally, between labels l_e and l_i, control is transferred to l_{start} if `Retry` is true; otherwise, the exception is pushed on top of the stack and re-thrown. The definition is:

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

	.try$\{$
	$\nabla_S\ (T_{S_1}, l_{start}, l_b, l_{retry}, l_c\ )$
$\{Q_n\}$	l_b : leave l_{next}
	$\}$
	catch $\mathit{System.Exception}$ $\{$
$\{Q_e \;\wedge\; \mathit{excV} \neq \mathit{null} \;\wedge\; s(0) = \mathit{excV}\}$	l_c : stloc $\mathit{last\_exception}$
$\{Q_e\ \}$	$\nabla_S\ (T_{S_2}, l_d, l_e, l_a, l_{exc}\ )$
$\{\mathit{Retry} \Rightarrow I_r \wedge \neg \mathit{Retry} \Rightarrow R_e\}$	l_e : ldloc Retry
$\{\mathit{Retry} \Rightarrow I_r \wedge \neg \mathit{Retry} \Rightarrow R_e \wedge\; s(0) = \mathit{Retry}\}$	l_f : brfalse l_h
$\{I_r\}$	l_g : br l_{start}
$\{R_e\ \}$	l_h : ldloc $\mathit{last\_exception}$
$\{R_e \;\wedge\; s(0) = \mathit{last\_exception}\}$	l_i : rethrow
	$\}$

6 Example

The PTC processes a list of Eiffel classes. Every class consists of a sequence of routines. Every routine consists of its pre- and postcondition and the source proof. The PTC generates two CIL types per Eiffel class: the interface and the implementation class. Then, for each routine, it translates the pre- and postcondition using the functions defined in Section 4. Finally it translates the source proof using the functions defined in Section 5.

```
 1 safe_division (x,y: INTEGER): INTEGER
     local
 3       z: INTEGER
     do
 5       { z=0 or z=1 }
         Result := x // (y+z)
 7       { zero and not_zero , z = 0 }
     ensure
 9       zero: y = 0 implies Result = x
         not_zero: y /= 0 implies Result = x // y
11   rescue
         { z=0 }
13       z := 1
         { z=1 , false }
15       Retry := true
         { Retry implies z=1 and not Retry implies false, false }
17   end
```

Fig. 2. Example of an Eiffel source proof.

Figure 2 and 3 illustrates the translation. Figure 2 presents the source proof. The example function implements an integer division, which always terminates normally. If the second operand is zero, it returns the first operand; otherwise the result is equal to the integer division $x//y$. Line 7 uses *zero and not_zero* to denote the properties expressed by the postcondition labeled with these names ($y = 0$ *implies Result* $= x$ *and* $y/ = 0$ *implies Result* $= x//y$). The bytecode proof uses the same convention. The exceptional postcondition of the last instruction of the `rescue` block and the exceptional postcondition of the routine are both false because the routine always terminates normally.

Figure 3 presents the bytecode proof. The generated bytecode for the body of the routine is enclosed in a try block (lines 01 to 07). Since the routine always terminates normally, the precondition of the instructions at labels 17 and 18 is false.

7 Soundness theorems

To be able to execute mobile code in a safe way, a soundness proof is required only for components of the trusted code base. Although PTCs are not part of the trusted code base, from the point of view of the code producer, the PTC should

	try {
$\{z = 0 \vee z = 1\}$	01 : **ldloc** x
$\{(z = 0 \vee z = 1) \wedge\ s(0) = x\}$	02 : **ldloc** y
$\{(z = 0 \vee z = 1) \wedge\ s(1) = x \wedge\ s(0) = y\}$	03 : **ldloc** z
$\{(z = 0 \vee z = 1) \wedge\ s(1) = x \wedge\ s(1) = y \wedge\ s(0) = z\}$	04 : **binop**$_{+}$
$\{(z = 0 \vee z = 1) \wedge\ s(1) = x \wedge\ s(0) = y + z\}$	05 : **binop**$_{//}$
$\{(z = 0 \vee z = 1) \wedge\ s(0) = x//(y + z)\}$	06 : **stloc** Result
$\{\mathit{zero\ and\ not\_zero}\}$	07 : **leave** 19
	}
	catch System.Exception {
$\{z = 0 \wedge\ \mathit{excV} \neq \mathit{null} \wedge s(0) = \mathit{excV}\}$	09 : **stloc** $\mathit{last\_exception}$
$\{z = 0\}$	10 : **ldc** 1
$\{z = 0 \wedge\ s(0) = 1\}$	11 : **stloc** z
$\{z = 1\}$	12 : **ldc** true
$\{z = 1 \wedge\ s(0) = \mathit{true}\}$	13 : **stloc** Retry
$\{\neg \mathit{Retry} \Rightarrow \mathit{false}\ \wedge \mathit{Retry} \Rightarrow (z = 1 \vee z = 0)\}$	14 : **ldloc** Retry
$\{\neg \mathit{Retry} \Rightarrow \mathit{false}\ \wedge$	
$\mathit{Retry} \Rightarrow (z = 1 \vee z = 0) \wedge\ s(0) = \mathit{Retry}\}$	15 : **brfalse** 17
$\{z = 1\ \vee z = 0\}$	16 : **br** 01
$\{\mathit{false}\}$	17 : **ldloc** $\mathit{last\_exception}$
$\{\mathit{false}\ \wedge\ s(0) = \mathit{last\_exception}\}$	18 : **rethrow**
	}
$\{\mathit{zero\ and\ not\_zero}\}$	19 : **ldloc** Result
$\{\mathit{zero\ and\ not\_zero} \wedge\ s(0) = \mathit{Result}\}$	20 : **ret**

Fig. 3. Bytecode proof generated by the PTC.

always generates valid proofs to avoid that the produced bytecode is rejected by the proof checker.

It is thus desirable to prove the soundness of both the proof translator and the specification translator. For the proof translator, soundness informally means that the translation produces valid bytecode proofs. It is not enough, however, to produce a valid proof, because the compiler could generate bytecode proofs where every precondition is false. The theorem states that if (1) we have a valid source proof for the instruction s_1, and (2) 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 (3) 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 (4) the exceptional postcondition in the source logic implies the precondition at the target label l_{exc} but considering the value stored in the stack of the bytecode, then every bytecode specification holds ($\vdash \{E_l\}\ I_l$). The theorem is the following:

Theorem 1

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

The soundness proof of the specification translator has been formalized and proved in Isabelle. First, we have defined evaluation functions from Eiffel expressions to values. $value_b$, $value_t$, $value_{exp}$, $value_{call}$ and $value_{arg}$ evaluate boolean expressions, Eiffel types, expressions, routine calls and arguments respectively. The theorem expresses: given two heaps and a state, 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

$$\begin{array}{l}
\forall b : boolExp,\ \ t : typeFunc,\ \ e : expr,\ \ c : CallRoutine,\ \ p : argument : \\
(wellF_b\ \ b) \Rightarrow (value_b\ \ b\ \ h_1\ \ h_2\ \ s) = ((\nabla_b\ \ b)\ \ h_1\ \ h_2\ \ s)\ \ \ and \\
(wellF_t\ \ t) \Rightarrow (value_t\ \ t\ \ h_1\ \ h_2\ \ s) = ((\nabla_t\ \ t)\ \ h_1\ \ h_2\ \ s)\ \ \ and \\
(wellF_{exp}\ \ e) \Rightarrow (value_{exp}\ \ e\ \ h_1\ \ h_2\ \ s) = ((\nabla_{exp}\ \ e)\ \ h_1\ \ h_2\ \ s)\ \ \ and \\
(wellF_{call}\ \ c) \Rightarrow (value_{call}\ \ c\ \ h_1\ \ h_2\ \ s) = ((\nabla_{call}\ \ c)\ \ h_1\ \ h_2\ \ s)\ \ \ and \\
(wellF_{arg}\ \ p) \Rightarrow (value_{arg}\ \ p\ \ h_1\ \ h_2\ \ s) = ((\nabla_{arg}\ \ p)\ \ h_1\ \ h_2\ \ s)
\end{array}$$

The full proofs can be found in our technical report [13]. 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 runs by induction on the syntactic structure of the expression and it is done in Isabelle.

8 Related work

Necula and Lee [12] 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, perform 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.

9 Conclusion

We have defined a proof-transforming compiler from a subset of Eiffel to CIL. The PTC allows us to develop certificates by interactively verifying source programs, and then translating the result to a bytecode proof. Since Eiffel supports multiple inheritance and CIL does not, we focused on the translation of contracts that refer to type information. We showed that our translation is sound, that is, it produces valid bytecode proofs. This translation can be adapted to other bytecode languages such as JVML. The main difference to CIL is the use of an exception table instead of .try and catch instructions as show in our previous work [11].

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

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. 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?
6. B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, second edition, 1997.
7. B. Meyer, P. Müller, and M. Nordio. A Hoare logic for a subset of Eiffel. Technical Report 559, ETH Zurich, 2007.
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 *Sixth International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2007)*, pages 39–46, 2007.
12. 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.
13. M. Nordio, P. Müller, and B. Meyer. Formalizing Proof-Transforming Compilation of Eiffel programs. Technical Report 587, ETH Zurich, 2008.
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. A Programming Logic for Sequential Java. In S. D. Swierstra, editor, *European Symposium on Programming Languages and Systems (ESOP'99)*, volume 1576 of *LNCS*, pages 162–176. Springer-Verlag, 1999.
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.