Integrating Proof-Transforming Compilation into EiffelStudio

Master Thesis

By: Manuel Hess
Supervised by: Martin Nordio
Prof. Bertrand Meyer

Student Number: 02-920-189
Contents

1 Introduction 2

2 Formalizing Proof-Transforming Compilation from Eiffel to CIL 6
2.1 Background on the logic for Eiffel and the logic for CIL Bytecode . 6
2.1.1 Logic for Eiffel .............................................. 6
2.1.2 Bytecode Logic ............................................. 8
2.2 Translation Functions ........................................... 10
2.3 Translation of Attributes ...................................... 12
2.4 Translation of Expressions .................................... 13
2.4.1 Basics on Translating Expressions .......................... 13
2.4.2 Non-strict Boolean Operators and then and or else ...... 14
2.5 Translation of Instructions ................................. 17
2.5.1 Assignment .................................................. 17
2.5.2 Conditional Instruction .................................... 17
2.5.3 Creation Instruction ....................................... 18
2.5.4 Read Attribute ............................................. 19
2.5.5 Write Attribute ............................................ 19

3 Architecture 20
3.1 Overview ..................................................... 20
3.2 Generating AST from XML Nodes ............................ 21
3.3 Implementing PTC ............................................. 22
3.3.1 Labeling ...................................................... 22
3.3.2 The Shift, Unshift & Replace Functions .................. 23
3.4 Translation of Attributes .................................... 25
3.5 Translation of Expressions ................................... 25
3.6 Translation of Instructions ................................... 25
3.7 Translation of other Rules ................................... 28
3.7.1 Translation of once Routines .............................. 28
3.7.2 Translation of local Variables ............................ 28
## CONTENTS

4 Interfacing Proof-Transforming compiler with EiffelStudio 30
  4.1 Adding a Tool in EiffelStudio .......................... 31
  4.2 Eiffel Code ........................................... 33
  4.3 PTC Feature Tool ....................................... 33
  4.4 Eiffel Proof ........................................... 35
    4.4.1 Highlight Conditions .............................. 35
    4.4.2 Context Action ..................................... 36
  4.5 CIL Proof ................................................ 36
    4.5.1 Bytecode with Proof .............................. 37
    4.5.2 Bytecode ........................................... 37
    4.5.3 Clear Panels ....................................... 37
    4.5.4 Syntax Tree ....................................... 38

5 Example 39
  5.1 Source Code ............................................ 39
  5.2 XML Input ............................................... 41
  5.3 Abstract Syntax Tree .................................. 44
  5.4 Eiffel Proof ........................................... 46
  5.5 CIL Proof ................................................ 49

6 Conclusion and Future Work 52

A Clusters and Class Overview 60

B Class Diagrams 65
Abstract

The execution of untrusted bytecode (such as mobile code) can produce unwanted behavior. A proof on the bytecode programs can be generated to ensure safe execution. Automatic techniques to generate proofs, such as certifying compilation, can only be used for a restricted set of properties such as type safety. In this thesis we present a proof- transforming compiler for a subset of Eiffel to .NET CIL. In particular we introduce new translations of the Eiffel specific keywords and then and or else. Furthermore we formalized the translation of classes routines and attributes. To show the feasibility of the approach we developed an PTC and a corresponding interface in EiffelStudio.
Acknowledgements

First I want to thank Martin Nordio for an interesting project and his constant assistance - it was a pleasure working with you. My gratitude goes also to Prof. Dr. Bertrand Meyer for giving me the opportunity to write this master thesis at the Chair of Software Engineering. Having the possibility to attend his lectures in the past years was an honour and a pleasure for me. Looking back on an instructive time at the ETH Chair of Software Engineering.
Chapter 1

Introduction

Today, developing modern state of the art object oriented software [14, 12] is complex. It is a great deal of team work and collaboration with many developers. The goal is always to produce correct code. However, it is not possible to provide this correct code. Mistakes are easily done.

With the introduction of development methodology and the reuse of software construction patterns [8] one can try to minimize the number of defects and to ensure a certain quality of the software. However the achievements in this sector are not enough. It is necessary to insure a testing methodology to minimize the number of errors in a software.

In this work we are interested in mobile code. In this sector a special quality is needed, as it is not possible to trust any entity involved in the process of software creation. The deployment over the internet provides an other lack of trusting the software.

There are different options to improve the quality of software. A first option consists of further improving the testing framework. Currently this is the most practiced approach to deliver software products. But testing will never be able to prove the absence of errors [4, 5, 6].

A second option is to apply software verification [19]: this technique can be applied either before the compilation on the source code level or after the compilation on the bytecode level.

Another option is the use of Proof-Carrying Code (PCC) [22]. PCC enables to execute mobile code in a safe way. The idea behind Proof-Carrying Code is that one attaches to the actual running code an proof which has to be easily checkable and its execution does not violate the safety policy of the system. However, nowadays it is only possible to generate proofs automatically for basic properties (type safety) [24].

Proof-Carrying Components (PCC) [22] are a form of trusted components, for which the high guarantee of quality is perhaps the strongest one possible. We have a mathematical proof which can be verified by a machine. The component satisfies specific properties, known as the contract for the component. These properties can
be more or less extensive; they might characterize all that’s interesting about the component’s behavior, or just some specific aspects, such as absence of “null-pointer dereferencing” or other run-time failures.

This Proof-Carrying Components can be automatically generated by using Proof-Transforming Compilers [24, 25, 20, 21]. PTC’s are similar to certifying compilers in PCC, but take a source proof as input and produce the bytecode proof (see Figure 1.1 for an overview). An important property of Proof-Transforming Compilers is that they do not have to be trusted [3, 2]. If the compiler produces a wrong specification or a wrong proof for a component, the proof checker will always reject any proof or specification which is not correct.

If the source and target language are similar, such a proof-transforming compilation is straightforward for example for JAVA [26, 29, 28]. But if there are large differences between the different concepts of the two languages the compilation will be difficult. For example while Eiffel [13, 17] supports multiple inheritance, CIL supports multiple inheritance only for interfaces.

The PTC can be separated into two modules:

- (1) a specification translator translating Eiffel contracts to CIL contracts
- (2) a proof translator translating Eiffel proofs to CIL proofs.

In this work, the PTC takes an Eiffel proof and produces the corresponding bytecode specification in CIL (Figure 1.1).

This project consists of the implementation of a PTC for Eiffel and the integration into EiffelStudio. The input of the PTC is a Eiffel program. To develop the proof, we use a Hoare -Style logic of a subset of Eiffel. This subset includes the following features:

- assignment instruction
- compositional instructions
- conditional instruction
- loop instruction
- rescue instruction
- retry instruction by variable
- object creation
- read and write attributes
- routine invocation
- once procedure and once functions

Further more we translate language independent rules such as the false axiom and the strength and weak rules.
Organization of the document

Chapter 2 focuses on the theoretical point of the thesis. We will present the foundations of translating Eiffel Code to CIL. As a reference I always refer to the Paper of Nordio et al. [16, 20, 21] which already developed a big number of translations. We investigate in depth the translating of the Eiffel language specific keywords and then and or else.

Chapter 3 focus on the implementation part of the Proof-Transforming Compiler. We will present the design choices we took and explain them in detail.

Chapter 4 focus on the new interface provided with the Proof-Transforming Compiler and explains the different views and features we have implemented to provide users an easy and helpful interface to analyze and develop new programs.
In chapter 5 we present an example on how the different intermediate translation steps look for a simple bank account example.

Finally, chapter 6 concludes the whole project and gives an outlook to further improvements and possible extensions to the Proof-Transforming Compiler.
Proof-transforming compilation has been formalized by Nordio et al [16, 20, 21]. In this formalization the proof-transforming compiler consists of two modules:

- a specification translator that translates Eiffel contracts to CIL contracts
- a proof translator that translates Eiffel proofs to CIL proofs.

This work focuses on the problems produced by the lack of features in CIL such as multiple inheritance and rescue blocks. However the features supported by Eiffel and not supported by CIL are not only the two we mentioned before.

In this chapter we formalize the translation of Eiffel expressions to CIL. In particular we introduce the definition of the translation of and then and or else expressions. We also extended the proof translation described by Nordio et al [24, 25] including the translation of classes routines and attributes.

The rest of this chapter is organized as follows. Section 2.1 gives some background on the logic we used for Eiffel and CIL. Section 2.2 introduces the translation functions for classes routines and attributes. Section 2.3 focus on the detailed translation of attributes from Eiffel to CIL. Section 2.4 gives some introduction in the translation of expression and then focuses on the translation of and then and or else expressions. Section 2.5 will show how instructions are translated using the introduced translation functions.

2.1 Background on the logic for Eiffel and the logic for CIL Bytecode

2.1.1 Logic for Eiffel

In this section we present the Eiffel subset used in this report and summarize the logic that is used for the verification of Eiffel programs.
2.1. Background on the logic for Eiffel and the logic for CIL Bytecode

The specification of a routine, or in general an instruction $I$, is defined as a Hoare triple of the form $\{P\} \; I \; \{Q_n, Q_e\}$, where $P$, $Q_n$, $Q_e$ are the deep core of Eiffel expressions extended with universal and existential quantifiers, and $I$ is a routine or an instruction. $P$ is the precondition which must hold before the execution of $I$. 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.

A specification $\{P\} \; I \; \{Q_n, Q_e\}$ defines the following refined partial correctness property: Suppose that $I$’s execution starts in a state which fulfills the precondition $P$, then one of the following must hold:

1. $I$ terminates normally in a state where $Q_n$ holds, or $I$ produces an exception and $Q_e$ holds

2. $I$ terminates due to errors or actions that are beyond the semantics of the programming language. (eg. memory allocation problems, interrupts)

3. $I$ runs forever.

To get an impression on how the logic looks like I will present three examples. First a simple assignment instruction then a conditional instruction and finally composition instruction. For all rules of the Eiffel logic see Nordio et al. [24]

**Assign Instruction**

The Hoare-style logic for an assignment instruction $x := e$ is the following:

$$\{P[e/x]\} \; x := e \; \{P, false\}$$

**Conditional Instruction**

Here we present the Conditional Instruction which models the if then else construct. It looks as follows:

$$\{P \land e\} \; s_1 \; \{Q_n, Q_e\}$$

$$\{P \land \neg e\} \; s_2 \; \{Q_n, Q_e\}$$

$$\{P\} \; \text{if } e \; \text{then } s_2 \; \text{else } s_2 \; \text{end} \; \{Q_n, Q_e\}$$
2.1. Background on the logic for Eiffel and the logic for CIL Bytecode

Composition Instruction

In the compositional statement the statement $s_1$ is executed first. Statement $s_2$ is executed if and only if $s_1$ has terminated in the normal state.

$$
\{P\} \quad s_1 \quad \{Q_n, Q_e\}
$$

$$
\{Q_n\} \quad s_2 \quad \{R_n, Q_e\}
$$

$$
\{P\} \quad s_1; s_2 \quad \{R_n, Q_e\}
$$

2.1.2 Bytecode Logic

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

The language of the bytecode is formed out of interfaces and classes. Each class has methods and fields. A method itself consists of a sequence of bytecode instructions. Bytecode instructions are labeled and operate on local variables, arguments the heap and mostly on the stack.

The bytecode language we use to express the logic is a variant of CIL with just little simplifications [15]. For local variables and arguments of routines we use the same instructions. In CIL one use an array of local variables. We use the name of the source variable. Also to simplify the translation to CIL we assume a boolean type in the bytecode language. The bytecode instructions and their informal description are the following:

- **ldc $c$**: pushes constant $c$ onto the stack
- **ldloc $v$**: pushes the value of a variable $v$ onto the stack
- **stloc $x$**: pops the top element of the stack and assigns it to the local variable $x$
- **binop**: removes the first two values from the stack and pushes the result of applying binop to these values
- **br $p$**: transfers control to the point $p$
- **brfalse $p$**: transfers control to the point $p$ if the topmost element of the stack is false and unconditionally pops it
- **rethrow**: takes the top value from the stack, assumed to be an exception, and rethrows it
- **leave $p$**: exit from the try or catch block to the point $p$
The bytecode logic we use is the logic developed by Bannwart and Müller [1].

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 \( I_l \).

Properties of methods are expressed by method specifications of the form \( \{ P \} \ T.mp \{ Q_n, Q_e \} \) 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 \( \{ P \} inst \{ Q \} \), where \( P, Q \) are first-order formulas and \( inst \) is a method body. The triple \( \{ P \} inst \{ Q \} \) expresses the following refined partial correctness property: if the execution of \( inst \) starts in a state satisfying \( P \), then

1. \( inst \) terminates in a state where \( Q \) holds
2. \( inst \) aborts due to errors or actions that are beyond the semantics of the programming language
3. \( inst \) runs forever

Assertions refer always 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 top element, element 1 the second topmost etc. The interpretation

\[ [E_l] : State \times Stack \rightarrow Value \]

for \( s \) is defined as follows:

\[ [s(0)]\langle S, (\sigma, v) \rangle = v \quad \text{and} \quad [s(i + 1)]\langle S, (\sigma, v) \rangle = [s(i)]\langle S, \sigma \rangle. \]

The functions shift and unshift express the substitutions that occur when one pushes or pops values onto and from the stack. The definitions of push and pop are the following:

\[ shift(E) = E[s(i + 1)/s(i) \forall i \in N] \]
\[ unshift = shift - 1 \]

\( shift^n \) denotes \( n \) consecutive applications of shift.

The rules for instructions have the following form:

\[ E_l \Rightarrow wp(I_l) \]
\[ \Delta \vdash \{ E_l \} l : I_l \]

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.

In the next table we show the definition of $wp(I_l)$:

<table>
<thead>
<tr>
<th>$I_l$</th>
<th>$wp(I_l)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldc $c$</td>
<td>unshift($E_{l+1}[c/s(0)]$)</td>
</tr>
<tr>
<td>ldlc $v$</td>
<td>unshift($E_{l+1}[v/s(0)]$)</td>
</tr>
<tr>
<td>stloc $x$</td>
<td>(shift($E_{l+1}$))[s(0)/$x$]</td>
</tr>
<tr>
<td>binop</td>
<td>(shift($E_{l+1}$))[s(1)ops(0)/s(1)]</td>
</tr>
<tr>
<td>br $p$</td>
<td>$E_p$</td>
</tr>
<tr>
<td>brfalse $p$</td>
<td>(s(0) ⇒ shift($E_{l+1}$)) ∧ (¬s(0) ⇒ shift($E_{p}$))</td>
</tr>
<tr>
<td>leave $p$</td>
<td>$E_p$</td>
</tr>
</tbody>
</table>

### 2.2 Translation Functions

The proof translator is based on six translation functions which translate the Eiffel proof, classes, routines, attributes, expressions and instructions. Before introducing these functions we define the data types used to model Eiffel and CIL proofs.

An Eiffel proof is a list of Eiffel class proof defined as follows:

**definition** $EiffelProof = list of ClassProof$

A class proof is a list of attributes followed by an list of routine proof. It’s definition is as follows:

**definition** $ClassProof = list of Attribute \ list of RoutineProof$

A routine proof is defined as a routine name, its arguments and its body, which is represented as a proof tree. The definition is as follows:

**definition** $RoutineProof = RoutineName \ list of Argument \ ProofTree$

A Bytecode proof is a list of CIL class proofs defined as follows:

**definition** $BytecodeProof = list of CILClassProof$

A CIL class proof is a list of attributes followed by an list of method proof. It’s definition is as follows:

**definition** $CILClassProof = list of CILAtribute \ list of CILMethodProof$

A method proof is defined as a method name, its arguments and its body, which is represented as a CIL proof tree. The definition is as follows:

**definition** $CILMethodProof = MethodName \ list of CILArgument \ CILProofTree$
2.2. Translation Functions

Signatures

The signatures of the six translation functions described above are defined as follows:

\[ \nabla : EiffelProof \to BytecodeProof \]

\[ \nabla_C : ClassProof \to CILClassProof \]

\[ \nabla_R : RoutineProof \to CILMethodProof \]

\[ \nabla_A : Attribute \to \text{list of CILMethodProof} \]

\[ \nabla_E : Precondition \times Expr. \times Postcond. \times Label \times Label \to BytecodeProof \]

\[ \nabla_S : ProofTree \times Label \times Label \times Label \times Label \to BytecodeProof \]

As the translation function for attributes \( \nabla_A \), expressions \( \nabla_E \) and instructions \( \nabla_S \) require a bit more explanation, they are explained separately in sections 2.2.1, 2.2.2, and 2.2.3.

Definition of translation function \( \nabla \)

Let \( p \) be an Eiffel proof defined as \( p = c_1 \ldots c_n \) where \( c_i \) is a class proof. The translation \( \nabla \) is defined as follows:

\[ \nabla(p) = \nabla_C(c_1) + \ldots + \nabla_C(c_n) \]
2.3 Translation of Attributes

Definition of translation function $\nabla_C$

Let $c$ be a class proof of the Eiffel class $c'$ defined as $c = a_1 \ldots a_n \ r_1 \ldots r_n$ where $a_i$ is an attribute and $r_i$ is a routine proof. The translation $\nabla_C$ is defined as follows:

$$\nabla_C(c) = \textit{.class } c' \{$$

$$\nabla_A(a_1)$$

$$\vdots$$

$$\nabla_A(a_n)$$

$$\nabla_R(r_1)$$

$$\vdots$$

$$\nabla_R(r_n)$$

$$\}$$

Definition of translation function $\nabla_R$

Let $r$ be a routine proof of the routine $r'$ with a body $b$ ($b$ is a proof tree) and be $a$ an argument of type $T$. The translation $\nabla_C$ is defined as follows:

$$\nabla_R(r) = \textit{.method } r' \{$$

$$\nabla_S(b, l_a, l_b, l_a, \phi)$$

$$l_b: \textit{ret}$$

$$\}$$

In the following sections we present the translation of attributes ($\nabla_A$), expression ($\nabla_E$) and instructions ($\nabla_S$). These translations are an contribution by this thesis.

2.3 Translation of Attributes

The Uniform Access Principle [14] states that all services offered by a class 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 $a$ of type $T$ defined in the class $AClass$, the followings methods and fields are created:

- In the interface class $Aclass$ : a method $T a(\cdot)$ and a method $set\cdot a(T)$ are defined.

- in implementation class Impl$AClass$ : a field public $T a$ is declared. Also the accessing functions get$\cdot a()$ and set$\cdot a(T)$ are declared.
To translate attributes we formalize the translation of attributes presented by Nordio et al [24, 25]. Let $a$ be an attribute of type $T$. The translation function is then defined as follows:

$$\nabla_A(a) =$$

```java
.field public class T a
.method public set_a(a_2: T){
    \{ a_2 \neq \text{null} \} L001 : ldloc a_2
    \{ a_2 \neq \text{null} \land s(0) = $(\text{instvar(this,T@a)}) \} L002 : ret
}
.method public get_a(): T{
    \{ a \neq \text{null} \} L003 : ldfld a
    \{ a \neq \text{null} \land s(0) = $(\text{instvar(this,T@a)}) \} L004 : ret
}
```

The state of 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 by Poetzsch-Heffter [27]. Following we present the definition of the operation we used to translate attributes and the read and write attribute

- **instvar**: $\text{Value} \times \text{FieldDeclId} \rightarrow \text{InstVar}$. It returns the instance variable lookup.

### 2.4 Translation of Expressions

The function $\nabla_E$ generates a bytecode proof from a source expression and a precondition for its evaluation. Because we introduce two new translation for binary boolean operators we have to extend the signature $\nabla_E$ from Nordio et al [25] by one more label. It has the following new signature:

$$\nabla_E : \text{Precondition} \times \text{Expression} \times \text{Postcond.} \times \text{Label} \times \text{Label} \rightarrow \text{Bytecode_Proof}$$

In $\nabla_E$ the first label is used as the starting label of the translation. The second label is used to indicate where to continue if a specific condition is reached. It is used like for example in the conditional translation to indicate where to continue if a condition is true. This new introduced label is used in the translation of the Eiffel specific keywords and then and or else. We explain these translation in detail in section 2.4.2

#### 2.4.1 Basics on Translating Expressions

Since we added a new argument to the translation function $\nabla_E$ we have to recapitulate the basic definition of the translation of constants, variables and binary
expressions. After that we present one of the contribution of this thesis: the translation of the boolean operators and then and or else.

**Constants**

Constants are translated using `ldc`. It means that we push the value of constant $c$ onto the stack

$$\nabla_E (Q \land \text{unshift}(P[c/s(0)])) \land c \land \text{shift}(Q) \land P, \ l_a, l_b) = \{Q \land \text{unshift}(P[c/s(0)])\} \ l_a : \text{ldc~} c$$

**Variables**

Variables are translated using `ldloc`. It means that we push the value of variable $v$ onto the stack

$$\nabla_E (Q \land \text{unshift}(P[x/s(0)])) \land x \land \text{shift}(Q) \land P, \ l_a, l_b) = \{Q \land \text{unshift}(P[x/s(0)])\} \ l_a : \text{ldloc~} v$$

**Expression $e_1$ op $e_2$**

In the translation of expressions, first the expression $e_1$ is evaluated and its result is pushed on the stack. Then the second expression is evaluated and also pushed onto the stack. After that the instruction for the operation is applied. The translations are defined as follows:

$$\nabla_E (Q \land \text{unshift}(P[e_1 \text{op} e_2/s(0)])) \land e_1 \text{op} e_2 \land \text{shift}(Q) \land P, \ l_a, l_d) =$$

$$\nabla_E (Q \land \text{unshift}(P[e_1 \text{op} e_2/s(0)])) \land e_1, \text{shift}(Q) \land P[s(0) \text{op} e_2/s(0)], \ l_a, l_d)$$

$$\nabla_E (\text{shift}(Q) \land P[s(0) \text{op} e_2/s(0)]), \ e_2, \text{shift}^2(Q) \land \text{shift} P[s(1) \text{op} s(0)/s(1), \ l_b, l_d)$$

$$\{\text{shift}^2(Q) \land \text{shift} (P[s(1) \text{op} s(0)/s(1)])\} \ l_c : \text{binop}$$

### 2.4.2 Non-strict Boolean Operators and then and or else

The semantics of `and then` and `or else` are defined as follows [14]:

- **a and then b** has value `false` if a has value `false` otherwise it has the value of b
- **a or else b** has value `true` if a has value `true` otherwise it has the value of b
The definitions seems to yield to the same semantics as the standard boolean operators \textit{and} and \textit{or}. The difference is what happens when \( b \) is not defined. In the case one uses the standard boolean operators the expression is undefined. But the above definitions may still give a result. If \( a \) is false \( a \textbf{ and then } b \) is false no matter what the result of \( b \) is. Similarly for \( a \textbf{ or else } b \) except that it hold for \( a \) is true.

A typical example to show the use of the boolean expression is:

\[(i \neq 0) \textbf{ and then }(j//i = k) \quad \text{(where } // \text{ is defined as integer division)}\]

which has the value false if \( i \) is equal to zero. If the expression had been written using \textbf{and} the second expression would be a division by zero if \( i = 0 \).

An expression \textbf{and then} always gives the same result as the corresponding expression using \textbf{and} if both are defined. But the \textbf{and then} may produce a value false in cases when the \textbf{and} form does not. The same holds for \textbf{or else} and the value true.

It is possible to express the non-strict operators through instructions in a language that does not include these operators. For example

\[(i \neq 0) \textbf{ and then }(j//i = k)\]

one can also write:

\[
\text{if } i = 0 \textbf{ then } b := \textit{false} \textbf{ else } b := (j//i = k)\textbf{end}
\]

The non-strict form is simpler but CIL does not support these operators. So we introduce as a contribution of this thesis two new translation functions which are defined as follows:

**Translation of and then**

First we use \( \nabla_E \) to evaluate the value of expression \( e_1 \). The result of \( e_1 \) is now on top of the stack. After that we have to check whether the result is \textit{true} or \textit{false}. When the expression \( e_1 \) evaluates to \textit{false} we know according to the definition of \textbf{and then} that the formula will never evaluate to \textit{true}. As \texttt{bfalse} pops the result and then jumps to the indicated label one has to load the result again. If \( e_1 \) has evaluated to \textit{true} then we have to translate \( e_2 \) by using \( \nabla_E \). After evaluating \( e_2 \) the result is on top of the stack. Regardless of the value we do not have to load a constant so we have to jump without popping the value from the stack which is done by the CIL instruction \texttt{br}.

We introduce one restriction to \( P \) which is that we use the boolean \textit{and} operator instead of the \textbf{and then}. Since the target language does not supports \textbf{and then} but according to the above definition of \textbf{and then} it will produce the same result. The formalized translation is the following:

\[
\nabla_E(Q \land \texttt{unshift}(P[e_1 \land e_2/s(0)], e_1 \textbf{ and then } e_2, \texttt{shift}(Q) \land P, l_a, l_\text{end})) =
\]
Informally one can say that this translation is sound. For example the postcondition of the first bytecode instruction \( bfalse \) is the precondition of the next instruction \( brtrue \). This also holds for the sequence of the other instructions. However it is necessary as future work to formally prove the soundness of this translation.

**Translation of or else**

First we use \( \nabla_E \) to evaluate the value of expression \( e_1 \). The result of \( e_1 \) is now on top of the stack. After that we have to check whether the result is \( true \) or \( false \). When the expression \( e_1 \) evaluates to \( true \) we now according to the definition of \( or \) \( else \) that the formula will never evaluate to \( false \). As \( brtrue \) pops the result and then jumps to the indicated label one has to load the result again. If \( e_1 \) has evaluated to \( false \) then we have to translate \( e_2 \) by using \( \nabla_E \). After evaluating \( e_2 \) the result is on top of the stack. Regardless of the value we do not have to load an constant so we have to jump without popping the value from the stack which is done by the CIL instruction \( br \).

\[
\nabla_E(Q \land unshift(P[e_1 \lor e_2/s(0)], e_1, shift(Q) \land P[s(0) \lor e_2/s(0)], l_a, l_{end})
\]

\[
\{shift(Q) \land P[s(0) \lor e_2/s(0)]\} \quad btrue \quad l_c
\]

\[
\nabla_E(Q \land unshift(P[(True \lor e_2)/s(0)], e_2, shift(Q) \land P[(True \lor s(0))/s(0)], l_c, l_{end})
\]

\[
\{shift(Q) \land P[s(0) \lor e_2/s(0)]\} \quad br \quad l_{end}
\]

\[
\{shift(Q) \land P[s(0) \lor e_2/s(0)]\} \quad ldc \quad true
\]
2.5 Translation of Instructions

The function $\nabla_S$ generates a bytecode proof from a source proof. The signature of these function looks as follows:

$$\nabla_S : \text{Proof\_Tree} \times \text{Label} \times \text{Label} \times \text{Label} \times \text{Label} \rightarrow \text{Bytecode\_Proof}$$

According to the introduction of the new label as an new argument to $\nabla_E$. We show the translation functions for an assignment instruction and a conditional instruction. The other translations can be found in Nordio et al [25] and can be adjusted accordingly. Then we present improved versions of the translation of creation instruction and read attribute which are contributions of this thesis.

2.5.1 Assignment

In the assignment instruction we first translate the expression $e$ by using $\nabla_E$. Then the result of the expression is stored to the variable $x$ using the stloc instruction. The definition of the assignment translation is the following:

$$\nabla_S \left( \{ P[e/x] \} \stackrel{x:=e}{\rightarrow} \{ P, \text{false, false} \}, l_{start}, l_{next}, l_{retry}, l_{exec} \right) =$$

$$\nabla_E (P[e/x], e, (\text{shift}(P[e/x]) \land s(0) = e), l_{start}, l_b)$$

$$\{ \text{shift}(P[e/x]) \land s(0) = e \} \quad l_b : \text{stloc} x$$

2.5.2 Conditional Instruction

As an other Example of how to use the new expression translation function $\nabla_E$ we present the conditional instruction. In this translation, the expression $e$ is translated using $\nabla_E$. 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 $\nabla_S$. 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:

$$T_{S_1} = \frac{\text{Tree}_1}{\{P \land e\} s_1 \{Q_n, Q_e\}}$$

$$T_{S_2} = \frac{\text{Tree}_2}{\{P \land \neg e\} s_2 \{Q_n, Q_e\}}$$
2.5. Translation of Instructions

The translation is as follows:

\[
\nabla_S \begin{pmatrix}
T_{S_1} & T_{S_2} \\
\text{if } e \text{ then } s_1 \\
\{P\} \text{ else } s_2
\end{pmatrix}, l_{\text{start}}, l_{\text{next}}, l_{\text{retry}}, l_{\text{exc}} = \nabla_E (P, e, (\text{shift}(P) \land s(0) = e), l_{\text{start}}, l_{b})
\]

\[
\{ \text{shift}(P) \land s(0) = e \}
\]

\[
l_b : \text{brtrue } l_e \\
\nabla_S (T_{S_2}, l_e, l_d, l_{\text{retry}}, l_{\text{exc}}) \\
l_d : \text{br } l_{\text{next}} \\
\nabla_S (T_{S_1}, l_e, l_{\text{next}}, l_{\text{retry}}, l_{\text{exc}})
\]

2.5.3 Creation Instruction

The main difference to the creation instruction translation in Nordio et al [24] is, that we simplify the creation feature make that it takes no arguments. We assume that the the creation only instantiate the object which means allocates it on the heap. The initialization of the object then is done by calling an init feature. With this simplification we can provide a much simpler translation of the creation instruction.

The rule is the following:

\[
\nabla_S \begin{pmatrix}
\{P\} \text{ T@make } \{Q_a, Q_e\} \\
\{P\} \text{ new}(\$T, T) \times (\$g < T > /\$e/p) \\
\{P\} \text{ new}(\$T, T) \times (\$g < T > /\$e/p) \text{ x } = \text{ new}(\$T, T)
\end{pmatrix}, l_{\text{start}}, l_{\text{next}}, l_{\text{retry}}, l_{\text{exc}} = \nabla_a : \text{newobj } root\_cluster.\text{Impl.T}
\]

\[
\{P[\text{new}(\$T, T), \$g < T > /\$e/p]\} \\
\{\text{shift}(P[\text{new}(\$T, T), \$g < T > /\$e/p]) \land s(0) = \text{new}(\$T, T)\} \\
\{P[\text{new}(\$T, T), \$g < T > /\$e/p] \land x = \text{new}(\$T, T)\} \\
\{\text{shift}(P[\text{new}(\$T, T), \$g < T > /\$e/p]) \land x = \text{new}(\$T, T) \land s(0) = x\}
\]

\[
l_a : \text{stloc } x \\
l_b : \text{idloc } x \\
l_c : \text{callvirt } T\text{@make}
\]
2.5. Translation of Instructions

2.5.4 Read Attribute

The translation of an read attribute is achieved by invoking the method \textit{get\_a()} where \textit{a} is the name of the attribute. We assume that all attribute names from the program are different. After execution of the method, the method then returns the field. \textit{S} is defined as the following precondition:

\[
S = \left\{ \left( y \neq \text{Void} \land P[\text{instvar}(y, S\text{@}a)/x]) \lor \begin{cases} y = \text{Void} \land Q_e \left[ \begin{aligned} & \text{void} /Nul\text{PExc} > /\$, \\
& \text{new}($/\$, Nul\text{PExc})/excV \end{aligned} \right] \end{cases} \right\}
\]

The translation is defined as follows:

\[
\nabla_S \left( \{S\} \ x := S\text{@}a\{P, Q_e\} \begin{array}{llll} l_{\text{start}}, l_{\text{next}}, l_{\text{retry}}, l_{\text{exc}} \end{array} \right) =
\]

\[
\{y = \text{instvar}(y, S\text{@}a))\}
\]

\[
\{s(0) = y \land \text{shift}(P[\text{instvar}(y, S\text{@}a)])\}
\]

\[
\{s(0) = \text{instvar}(y, S\text{@}a) \land \text{shift}(P[\text{instvar}(y, S\text{@}a)])\}
\]

2.5.5 Write Attribute

To write an attribute, the method \textit{set\_a} is used. The placeholder \textit{a} is the name of the attribute to write. First the object and the according expression is pushed on the top of the stack. Then the method \textit{set\_a} is called. Let \textit{S} be the following precondition

\[
S = \left\{ \left( y \neq \text{Void} \land P[\text{void} < \text{instvar}(y, S\text{@}a) := e > /\$]) \lor \right. \right. \begin{cases} y = \text{Void} \land Q_e \left[ \begin{aligned} & \text{void} /Nul\text{PExc} > /\$, \\
& \text{new}($/\$, Nul\text{PExc})/excV \end{aligned} \right] \end{cases} \right\}
\]

The translation \( \nabla_S \) is defined as follows:

\[
\nabla_S \left( \{S\} y, S\text{@}a := e \{P, Q_e\} \begin{array}{llll} l_{\text{start}}, l_{\text{next}}, l_{\text{retry}}, l_{\text{exc}} \end{array} \right) =
\]

\[
\{y \neq \text{void} \land P[\text{void} < \text{iv}(y, S\text{@}a) := e > /\$]) \lor \begin{cases} y = \text{Void} \land Q_e \left[ \begin{aligned} & \text{void} /Nul\text{PExc} > /\$, \\
& \text{new}($/\$, Nul\text{PExc})/excV \end{aligned} \right] \end{cases} \right\}
\]

\[
\nabla_E (\{s(0) = y \text{shift}(P[\text{void} < \text{iv}(y, S\text{@}a) := e > /\$])\}, e, \{s(1) = y \land s(0) = e \land \text{shift}^2(P[\text{void} < \text{iv}(y, S\text{@}a) := e > /\$])\}, l_b, l_c)
\]

\[
\{s(1) = y \land s(0) = e \land \text{shift}^2(P[\text{void} < \text{iv}(y, S\text{@}a) := e > /\$])\}
\]

\[
\{s(1) = y \land s(0) = \text{callvirt } S\text{@}set\_a \}
\]
Chapter 3

Architecture

3.1 Overview

Figure 3.1 shows the general idea and the overall flow of the compilation process of the proof-transforming compiler: the developer provides a proof for an Eiffel class in XML.

Once the proof is developed and encoded in XML format, the PTC is able to process the proof. The compiler processes the input and creates internally a tree representing the proof’s structure. The internal tree is visited and changed according to the needs of the proof translator. After that the translation to CIL bytecode [9] is applied.

The XML Parser and the internal representation of the XML Nodes were implemented by Karahan [11]. In the next sections we will focus on the AST, the Proof Translator and the AST Bytecode.
3.2. Generating AST from XML Nodes

To compile an Eiffel Source several actions take place:

• transform the XML Nodes into an AST

• per every class proof translate its attributes and methods

• per every method translate its instructions.

These actions are explained in detail in the following sections.

The source code for the compiler and its corresponding interface can be found on Origo. http://paco.origo.ethz.ch/

3.2 Generating AST from XML Nodes

In the first part of the compilation process the parser provides an abstract syntax tree (AST) from the XML-Input file. To get an deeper look a how the concepts are and how in detail this is implemented see Karahan [11]. What we get after the first part is the following picture (Figure 3.2):

```eiffel
feature (NONE) -- attributes
attr: TUPLE [id : LIST (XML_ATTRIBUTE); -- ast-id
       precondition : LIST (STR_PRECONDITION); -- proof-item
       postcondition : LIST (CPX_POSTCONDITION); -- proof-item
       expression : LIST (STR_EXPRESSION);
       then_part : LIST (REF_THEN_PART);
       else_part : LIST (REF_ELSE_PART)]
```

Figure 3.2: XML node data structure

This represents nearly one to one the XML file which is in fact to detailed in the type information provided. Also the way it is represented as TUPLE with LISTs is not appropriate for our compilation phase. What we want to have is an easy access to the abstract syntax tree without lists and tuples. Especially in this example there exists only one expression, then_part and else_part. There is no need to have the model of lists.

Another important change is done in the types of the nodes. As we have in the XML node types such as REF_THEN_PART and REF_ELSE_PART we do not need this detailed type information. It is enough to have the general information that the if part has the type of a AST_PROOF_ITEM. This explicit typing realized by assignment attempts allows us to apply a easy visitor pattern to walk through the tree and translate the whole proof tree. We want something like showed in Figure 3.3:

This target is achieved by traversing the abstract syntax tree a second time. We then apply an special method xml2ast which makes these transformation. The transformation is done by using assignment attempts like then_part? = attr.then_part.first.
This means that we access the first element of the list _then_part_ in the tuple _attr_. When all the necessary attributes are converted a flag _parsing_finished_ is set to true.

The method _xml2ast_ is an empty implementation in _AST_PROOF_ITEM_. So in each descendants (see Appendix B) of _AST_PROOF_ITEM_ one has to redefine this feature in the appropriate way. If it is not redefined nothing will be done.

The invariant of each effected class is that all the attributes are not void after the _parsing_finished_ flag is set to true

\[
\text{invariant : converted : parsing\_finished } \implies
\text{(precondition} \neq \text{Void} \land \text{postcondition} \neq \text{Void}\land \\
\text{expression} \neq \text{Void} \land \text{then\_part} \neq \text{Void} \land \text{else\_part})
\]

### 3.3 Implementing PTC

In the next sections we will present some details about the practical implementation of the proof transforming compiler. Section 3.3.1 gives detailed information about how we managed the labeling of the CIL bytecode. Section 3.3.2 gives information about how to implement the theoretical shift and unshift function introduced in Section 2.1.2.

#### 3.3.1 Labeling

Labeling is an important part of the translation. There has to be an continues increasing number for each new CIL instruction. The class responsible for the labeling is _LABELGENERATOR_. There exists only one instance of it through the whole translation phase (_make_ is a once feature). The function always returns a label as a STRING of the form L00x. Where x is an value (integer) which starts at 1 and increases by every new CIL instruction.

Internally there is one counter which takes track of the current label enumeration and can be accessed through the following interface:

```
feature -- AST attributes after parsing

 precondition: STR\_PRECONDITION
 postcondition: CPX\_POSTCONDITION
 expression: VIS\_CONTRACT --AST\_BOOLEXP
 then\_part: AST\_PROOF\_ITEM
 else\_part: AST\_PROOF\_ITEM

parsing\_finished: BOOLEAN
```

Figure 3.3: AST data structure used for translation
3.3. Implementing PTC

- **succ**: returns a label with value current counter and increases the counter by one.

- **next**: returns a label with value current counter plus 1 and does not increase the counter.

- **prev**: returns a label with value current counter minus 1 and does not change the counter.

- **reset**: sets the internal counter to zero.

The features `next` and `prev` are especially handy if one translates instruction like if then else. The counting continues through out the whole translation of this instruction. Nevertheless one does not know how many CIL instruction for a translation of the expression part or the else part are needed. Since we know at the end of each translation what the value of the current label is we can then adjust the Labels used in the CIL instructions `br` or `brfalse`. It is possible to set the label where the control flow is transferred next. So it is possible that one can at the right place modify the labels of an `br` or `brfalse` instruction by invoke the method `set_branch_label(l: STRING)`.

As a reference implementation please have a look at the PIT_IF_PROOF at the `translate` method. One can modify the label on a `br` or `brfalse` instance by invoking `set_branch_label(l: STRING)`.

### 3.3.2 The Shift, Unshift & Replace Functions

For compiling the CIL with the proof one has to reference to the current stack. This is done by printing $s(0) = x \land s(1) = 3$. It can be read as that the stack looks as following:

![The Stack](image)

Each time one pushes or pop's something from the stack, it has to be adjusted. It also has to be possible to replace an entry at a specific position in the stack. The stack notation is implanted as an extension to the abstract syntax tree of the contracts and can be found in the class VIS_STACK. The abstract base class of the Contract AST: AST_VISITABLE has two `deferred` features:
3.3. Implementing PTC

- **shift_unshift**: increments every VIS_STACK instance by one respectively by minus one. Technically one goes through the whole AST and every time one reaches an VIS_STACK instance the corresponding counter of VIS_Stack is incremented or decremented. This is visualized by the following pictures:

![shift.png](attachment:image1.png)

Figure 3.5: The shift function

![unshift.png](attachment:image2.png)

Figure 3.6: The unshift function

- **replace(i: INTEGER, e: AST_VISITABLE)**: it implements the replacement of instruction in the formalized translation s(1)/y. This is read as replace whatever is on position one on the stack by y.

Technically each visitable class implements the replace function. It asks if the expression to the left or right is of type VIS_STACK and if the counter is the same as the argument i. If yes then replace the whole expression by e. Else invoke the replace method on the next AST node. The stack looks after an replacement s(1)/y as follows:

![replace.png](attachment:image3.png)

Figure 3.7: The replace function
Each descendant of AST_VISITABLE has to implement its own feature `shift_unshift` and `replace`. For an complete overview see Appendix B.

As an reference implementation please have a look at the feature `translate_expression` in class VIS_EQ.

### 3.4 Translation of Attributes

As presented in Section 2.2.6, attributes are also a special case for the translation. Every attribute is translated to two methods: `set_attribute` and `get_attribute`. This translation is done first every time one translates a new class. The corresponding implemented feature can be found in class CPX_CLASS_PROOF.

In feature `translate` is a code section which loops through all attributes and calls on each of them the feature `translate_attributes` which then generates the setter and getters for the specific attribute.

### 3.5 Translation of Expressions

The translation of the expression are done using the $\nabla_{E}$ function defined in Section 2.4. In the implementation of the PTC we have defined a deferred function `translate_expression` in the deferred class VIS_CONTRACT. All the descendant classes have to implement this translation function. The invocation is done by a visitor which invokes the feature `translate_expression` on each visited node in the abstract syntax tree. For an overview of the descendants of VIS_CONTRACT go to Appendix B.

Table 3.5 gives a quick overview which operators for expressions are implemented and in which class to find the corresponding translation function.

### 3.6 Translation of Instructions

The translation of the instructions are done using the $\nabla_{S}$ function defined in Section 2.5. In the implementation of the PTC we have defined a deferred function `translate` in the deferred class AST_PROOF_ITEM. All the descendent classes have to implement this translation function. The invocation is done by a visitor which invokes the feature `translate` on each visited node in the abstract syntax tree. If an instruction contains an expression for example pre - and postconditions or simple calculations one has to invoke the `translate_expression` feature on this VIS_CONTRACT node. For an example of translating an if then else instruction go to feature `translate` in class PIT_IF_PROOF. For an overview of the descendants of AST_PROOF_ITEM go to Appendix B.

Table 3.6 gives a quick overview which instructions are implemented and in which class to find the corresponding translation function according to the rules defined in Nordio et al [24, 25] and Section 2.5.
<table>
<thead>
<tr>
<th>operator</th>
<th>corresponding translating class</th>
</tr>
</thead>
<tbody>
<tr>
<td>stack</td>
<td>VIS_STACK</td>
</tr>
<tr>
<td>allocation</td>
<td>VIS_ALLOC</td>
</tr>
<tr>
<td>store</td>
<td>VIS_STORE</td>
</tr>
<tr>
<td>stack</td>
<td>VIS_STACK</td>
</tr>
<tr>
<td>addition</td>
<td>VIS_ADD</td>
</tr>
<tr>
<td>subtraction</td>
<td>VIS_SUB</td>
</tr>
<tr>
<td>multiplication</td>
<td>VIS_MUL</td>
</tr>
<tr>
<td>division</td>
<td>VIS_DIV</td>
</tr>
<tr>
<td>logical and</td>
<td>VIS_AND</td>
</tr>
<tr>
<td>logical or</td>
<td>VIS_OR</td>
</tr>
<tr>
<td>logical not</td>
<td>VIS_NOT</td>
</tr>
<tr>
<td>logical and then</td>
<td>VIS_ANDTHEN</td>
</tr>
<tr>
<td>logical or else</td>
<td>VIS_ORElse</td>
</tr>
<tr>
<td>false axiom</td>
<td>VIS_FALSE</td>
</tr>
<tr>
<td>true axiom</td>
<td>VIS_TRUE</td>
</tr>
<tr>
<td>less then</td>
<td>VIS_LT</td>
</tr>
<tr>
<td>less then or equal</td>
<td>VIS_GTE</td>
</tr>
<tr>
<td>greater then</td>
<td>VIS_GT</td>
</tr>
<tr>
<td>greater then or equal</td>
<td>VIS_GTE</td>
</tr>
<tr>
<td>equal</td>
<td>VIS_EQ</td>
</tr>
<tr>
<td>not equal</td>
<td>VIS_NEQ</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 3.1: Translation of Expressions
<table>
<thead>
<tr>
<th>rule</th>
<th>corresponding translating class</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>PIT_ASSIGNMENT</td>
</tr>
<tr>
<td>conditional</td>
<td>PIT_IF_PROOF</td>
</tr>
<tr>
<td>compositional</td>
<td>PIT_COMPOSITION_PROOF</td>
</tr>
<tr>
<td>loop</td>
<td>PIT_LOOP_PROOF</td>
</tr>
<tr>
<td>rescue</td>
<td>PIT_RESCUE_PROOF</td>
</tr>
<tr>
<td>retry</td>
<td>PIT_RESCUE_PROOF</td>
</tr>
<tr>
<td>check</td>
<td>PIT_CHECK_PROOF</td>
</tr>
<tr>
<td>creation of objects</td>
<td>PIT_CREATE_PROOF</td>
</tr>
<tr>
<td>invocation</td>
<td>PIT_INVOCATION_PROOF</td>
</tr>
<tr>
<td>read attributes</td>
<td>PIT_READ_PROOF</td>
</tr>
<tr>
<td>write attributes</td>
<td>PIT_WRITE_PROOF</td>
</tr>
<tr>
<td>once procedure</td>
<td>CPX_ROUTINE_PROOF</td>
</tr>
<tr>
<td>once function</td>
<td>CPX_ROUTINE_PROOF</td>
</tr>
<tr>
<td>strength</td>
<td>PIT_STRONG_PROOF</td>
</tr>
<tr>
<td>weak</td>
<td>PIT_WEAK_PROOF</td>
</tr>
<tr>
<td>false axiom</td>
<td>PIT_FALSE_AXIOM</td>
</tr>
<tr>
<td>conjunction</td>
<td>PIT_CONJUNCTION_PROOF</td>
</tr>
<tr>
<td>disjunction</td>
<td>PIT_DISJUNCTION_PROOF</td>
</tr>
<tr>
<td>invariant</td>
<td>STD_INVARIANT_RULE</td>
</tr>
<tr>
<td>substitution</td>
<td>STD_SUBSTITUTION_RULE</td>
</tr>
</tbody>
</table>

Table 3.2: Translation of Instructions
3.7 Translation of other Rules

As once routines and local variables also are also treated specially in the implementation of the PTC we present this translations here. Section 3.7.1 presents the translation of once routines. Once routines are executed only once. Then they store first the compute value and on each invocation this stored value is used or returned. Section 3.7.21 presents the translation of local variables which have to be initialized by the beginning of the feature.

3.7.1 Translation of once Routines

A once routine is a routine that is executed only once. There are two different kind of once routines: Once procedures and once functions. The first invocation of an once procedure executes the body of the procedure. However, the remaining executions do nothing. Once functions return a value. The returned value of once functions is always the same value (which was obtained in the first invocation). More in detail analysis on once routines in Nordio et al [24, 25]. According to this behavior of this routine types Special care has to be taken at the translation of once routines.

The distinction of which translation has to be applied happens in the method translate in class CPX_ROUTINE_PROOF. A special boolean flag set in the source XML indicates if it is an once routine or an normal one. The distinction between a procedure and a function is achieved by looking if there is an return value or not. According to the results the appropriate translation method is invoked.

3.7.2 Translation of local Variables

In Eiffel there is a default initialization on local variables. Since these is not supported by CIL one has to provide also an initialization on bytecode level.

This is done by an special method translate_local in the class CPX_ROUTINE_PROOF. The initialization is as follows:

- INTEGER: initializing value is zero
- BOOLEAN: initializing value is false
- other variables: initializing value is void

With this explicit initialisation of local variables it is guaranteed that Eiffel code such as

local_example: INTEGER is
  -- returns 5 in a complicated way
local
  zero: INTEGER
do
  Result := zero + 5
end

is translated to:

{True} IL001: ldc 0
{(True and (s(0)=0))} IL002: stloc zero
{(True and (s(0)=0))} IL002: ldloc v
{(True and (s(0)=0))} IL004: ldc 5
{(True and (s(1) + s(0) = 0+5))} IL005: add
...

Chapter 4

Interfacing Proof-Transforming compiler with EiffelStudio

To provide an interface to the proof-transforming compiler we decided to integrate it into EiffelStudio. We reused the features and editors provided by EiffelStudio and adapted them according to our needs. The result are three different new tools to provide a user friendly interface which supports developing and controlling the generated proofs.

A schematic overview is given in Figure 4.1.

![Diagram of Interfaces for PTC]

Figure 4.1: Interfaces for PTC

The default screen (Figure 4.2) is divided into four different sections. In the upper left corner one finds the Eiffel Code window which is described in Section 4.2. In the upper right corner the PTC Feature Tool is located. Its in detail explanation can be found in section 4.3. In the lower left corner the one finds the Eiffel Proof...
4.1 Adding a Tool in EiffelStudio

To add a new tool to EiffelStudio [7] one has to do several things. First one has to create a new Tool descriptor. It is basically a brief description of the new tool. All tool descriptors inherit from ES_TOOL taking a generic parameter representing the tools user interface panel. Figure 4.3 describes the first few lines of each new tool descriptor.

One of the most important function of the descriptor is create_tool. It’s used to instantiate the tools user interface panel, which will be requested when the tool is shown, either in full (that tool is visible) or in part (tabbed or hidden by another tool) in place in the EiffelStudio IDE.

As implementation reference one can have a look at the classes CIL_TOOL, EIFFEL_PROOF_TOOL or PTC_FEATURE_TOOL.

To each tool one need the corresponding panel which is the actual GUI and
Adding a Tool in EiffelStudio

visible part part of the tool. In this class there is also all the code which reacts on user input like mouse clicks or other events. Figure 4.4 shows the first few lines of each tool panel.

In the feature make on can initialize the new tool and register it to the different observers. Its recommended to separate the building of the interface and the actual registration at the observers in different features. As implementation reference one can have a look at the classes CIL_TOOL_PANEL, EIFFEL_PROOF_TOOL_PANEL or PTC_FEATURE_TOOL_PANEL.

One of the final things to do is to add the tool to the View > Tool menu in the EiffelStudio IDE. In the class EB_DEVELOPMENT_WINDOW_MENU_BUILDER in EiffelStudio locate the feature toollist_menu. A quick look at the code reveals that one has to add insert_show_tool_menu_item(Result, {*ClassnameoftheTOOL*})

class
  EIFFEL_PROOF_TOOL

inherit
  ES_TOOL [EIFFEL_PROOF_TOOL_PANEL]

create {NONE}
  default_create

Figure 4.3: Header of a TOOL descriptor

class
  EIFFEL_PROOF_TOOL_PANEL

inherit
  EB_TOOL

redefine
  make
  end

create
  make

feature {NONE} -- Initialization

  make (a_tool: EB_DEVELOPMENT_WINDOW;
  a_desc: like tool_descriptor) is
    -- Create a new external tool.

Figure 4.4: Header of a TOOL_PANEL implementation
at the appropriate position.

To get an easy reference to the actual tool on the GUI one has to register the new tool in the class EB.DEVELOPMENT.WINDOW.TOOLS. This allows to statically reference the tool. Figure 4.5 shows the piece of code which has to be added:

cil_tool: EIFFEL_PROOF_TOOL_PANEL
  -- EIFFEL_PROOF tool
  require
    not_is_recycled: not is_recycled
  do
    Result ?= develop_window.shell_tools.tool
      ({EIFFEL_PROOF_TOOL}).panel
  ensure
    result_attached: Result /= Void
  end

Figure 4.5: Registration of a tool

With this registration it is possible to access the tool panel in the code with develop_window.tools.cil_tool.

All the three new tools are implemented in this way. Section 4.3 shows the details of the PTC Feature Tool. Section 4.4 explains the Eiffel Proof Tool and Section 4.5 shows the details of the CIL Tool.

4.2 Eiffel Code

This is the same window as it is for normal developing Eiffel programs. It behaves exactly the Eiffel Studio build in Editor. Figure 4.6 shows the source code for a simple bank account example in the original Eiffel editor.

4.3 PTC Feature Tool

The new PTC Feature Tool shown in Figure 4.7 supports the easy navigation through complex and long programs. It is intended to behave exactly the same way as the feature navigation of the EiffelStudio Feature does.

If one clicks on an feature or attribute the code in the Eiffel source, the Eiffel proof tool and CIL tool will jump to the beginning of the selected feature or attribute. The basic functionalities of the PTC feature tool are inherited from the original feature tool of EiffelStudio. The nice thing about using the the corresponding EiffelStudio tool is that we can access the internal representation of the Eiffel source. This allows us to display the features grouped which is really helpful to navigate through large
source files. For example as shown in Figure 4.7 the grouping into initialization, access and implementation.

The corresponding feature which handles the event on which line in the editor to jump, is solved by implementing the call back function go_to_line. As an example on this consolidate in class EIFFE_PROOF_TOOL_PANEL feature go_to_line.
4.4 Eiffel Proof

The representation of the source code in the Eiffel Proof Tool (Figure 4.8) is slightly different than the original Eiffel source code. It is used to visualize the new added pre-and postconditions for every instruction. Here the keyword condition is used as the postcondition of Instruction $I_n$ is also the precondition of instruction $I_{n+1}$.

It is also a visualization of the generated XML nodes from the XML input files. It follows the convention that all attributes of the class have to be defined before the routines.

![Eiffel Proof Tool](image)

Figure 4.8: Eiffel Proof Tool

4.4.1 Highlight Conditions

A special feature of this tool the highlight condition Feature (Figure 4.9) which can by activated by pressing the corresponding button above the editor pane. Its goal is to focus only on the conditions of the eiffel proof. This means on the pre -and postcondition of every feature and as well on the conditions for every instruction. This is done by highlighting the corresponding lines in the editor pane. The require and ensure clauses on the CIL editor are highlighted as well.

The corresponding agent implementation can be found in feature highlight_condition in class EIFFEL_PROOF_TOOL_PANEL. The feature highlight_condition works in such a way that it first divides the text into lines and then loops through the lines and looks for the keywords require, ensure and condition. Each line number
which has to be highlighted is stored in an array of integers. This array is the passed to a feature `highlight` which invokes the appropriate feature to highlight the editor pane. Figure 4.9 shows the result.

![Figure 4.9: Highlighting conditions](image)

### 4.4.2 Context Action

To simplify the navigation in the eiffel proof and the CIL proof we implemented a context action which navigates the user to the corresponding compiled CIL method. (See Figure 4.10) The behavior of the context action is as follows:

- right mouse click on a feature name or somewhere between in the feature will pop up a context menu which contains the line which was selected.

- a left mouse click on the context menu entry with the selected lines forces the CIL editor to jump to the corresponding beginning of the compiled method.

The reference implementation can be found in the features `on_right_click`, `context_menu_handler` and the `agent jump_to_cil` in the class `EIFFEL_PROOF_TOOL_PANEL`. First one has to implement the callback function `context_menu_handler`. There one can add the corresponding menu entries and define which agent is called if this entry is chosen by the user. The feature `go_to_cil_proof` is invoked. This features determines in which method the right click occurred and then searches for the corresponding method either in the cil code or the eiffel proof code.

The context action is also available in the CIL Tool and works the same way vica versa.

### 4.5 CIL Proof

This window is used to present the compiled CIL code with its proof in an nice way. (Figure 4.11) Left in the high green color the proof is written down. Each line can be derived by the previous one. The Syntax is derived from the CIL Standard.

The CIL Code consists of an Label and an CIL Instruction. Right to the proof
always follows an label. After that the actual CIL Instruction follows.

Above the Editor there are four different Buttons which are described in the following sections.

4.5.1 Bytecode with Proof

Press this button to show in the CIL proof window the proof and the bytecode instructions

4.5.2 Bytecode

Press this button to show only the bytecode instructions without the corresponding proof. These view is useful if one only need to have a look at the CIL compilation.

4.5.3 Clear Panels

With this button one can clear both Windows the CIL window and the Eiffel Proof window.

![Image of Eiffel Proof Tool conditions]

Figure 4.10: Context Action in Eiffel Proof Tool conditions
4.5.4 Syntax Tree

With these button the intermediate abstract syntax tree after parsing the XML input. These view is especially useful if an parsing error occurs. This view gives a hint where to find the parsing error.

```
.method public void make (string n) {
    require
        {n} != Null

    code
        {{n} != Null} and (n=n)}
        IL001: ldloc n

        {{n} != Null} and (n=0)}
        IL002: stloc name

        {{{name} = n} and (0=0)}}
        IL003: ldc 0

        {{{name} = n} and (0=0)}}
        IL004: stloc balance

    ensure
        {{{name} = n} and (balance=0)}

}
Chapter 5

Example

As an Example of how the different steps look like I will present an example of an simple bank account. First we have normal Eiffel source code. From this source code one has to derive an XML file which then is used to present an abstract form of the Source code. Also from the intermediate XML File and from the resulting abstract syntax tree the translation is applied and produces the CIL Code.

In the following sections these steps are provided in detail

5.1 Source Code

Here we show the code of an simple bank account example. It contains the feature deposit and withdraw which increases or decreases the money of the bank account. Every feature is implemented with its appropriate pre -and postconditions such as that one can not withdraw money if the balance is less then the amount to withdraw (see require clause offeature withdraw). The class BANK ACCOUNT has a creation procedure make which initializes the account with zero and assigns it with a name. The attributes name and balance of the class models the owner of the account and the current amount of money.

Following one find the complete code in Eiffel:

indexing

   description: "Object that represents a bank account"

class
    BANK ACCOUNT

create make

feature {NONE} -- Initialization
    make (n: STRING) is
       -- create a bank account with name 'n'
require
  n /= Void
do
  name := n
  balance := 0
ensure
  name = n
  balance = 0
end

feature -- access

deposit (v: INTEGER) is
  -- deposit 'v' in the current bank account
require
  v > 0
do
  balance := balance + v
ensure
  balance = old balance + v
end

withdraw (v: INTEGER) is
  -- withdraw 'v' in the current bank account
require
  balance > v
do
  balance := balance - v
ensure
  balance = old balance - v
end

feature -- implementation
balance: INTEGER
name: STRING

invariant
  name_not_void: name /= Void
  balance_positive: balance >= 0
end
5.2 XML Input

This is the XML representation of the above Eiffel code. The whole class is represented as a class proof with the list of attributes balance and name. Then the list of routine proofs follow. Attributes are translated as:

```xml
<feature>
  <name>balance</name>
  <type>INTEGER</type>
</feature>
```

The routine tag contains the name followed by the argument tag which models the arguments for a routine. The pre- and postconditions of a routine for example routine withdraw (precondition: balance > v postcondition: balance = oldbalance−v) are represented as follows:

```xml
<require>balance{GT}v</require>
<ensure>balance{=}old{SP}balance{−}v</ensure>
```

Each instruction is enclosed by its proof rule tag. For example assignments are enclosed by the `<assignment−proofID = "ass−0" > tag. The glue between the instruction build the composition proof which takes two references to indicate which instruction follows on an other.

Following one find the complete code of the XML input file for the bank account example:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<eiffel-proof xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="/xsd/eiffel-proof.xsd"

<class-proof>
  <name>bank_account</name>

  <feature>
    <name>balance</name>
    <type>INTEGER</type>
  </feature>

  <feature>
    <name>name</name>
    <type>STRING</type>
  </feature>

  <routine-proof>
```
<name>make</name>

<argument>
  <name>n</name>
  <type>string</type>
</argument>

<require>n(!)=Void</require>
<ensure>(name(=)n{and}(balance(=)0)</ensure>

<composition-proof ID="comp-0">
  <precondition>n(!)=Void</precondition>
  <postcondition>
    <normal>(name(=)n{and}(balance(=)0)</normal>
  </postcondition>
  <s1 ref="ass-0" />
  <s2 ref="ass-1" />
</composition-proof>

<assignment-proof ID="ass-0">
  <precondition>n(!)=Void</precondition>
  <postcondition>
    <normal>name(=)n</normal>
  </postcondition>
  <target>name</target>
  <expression>n</expression>
</assignment-proof>

<assignment-proof ID="ass-1">
  <precondition>name(=)n</precondition>
  <postcondition>
    <normal>(name(=)n{and}(balance(=)0)</normal>
  </postcondition>
  <target>balance</target>
  <expression>0</expression>
</assignment-proof>

</routine-proof>

<routine-proof>
  <name>deposit</name>

  <argument>
    <name>v</name>
    <type>integer</type>
  </argument>

  <require>v(!)=Void</require>
  <ensure>(v(=)v{and}(balance(=)+v))</ensure>

  <composition-proof ID="comp-1">
    <precondition>v(!)=Void</precondition>
    <postcondition>
      <normal>(v(=)v{and}(balance(=)+v))</normal>
    </postcondition>
    <s1 ref="ass-2" />
    <s2 ref="ass-3" />
  </composition-proof>

  <assignment-proof ID="ass-2">
    <precondition>v(!)=Void</precondition>
    <postcondition>
      <normal>v</normal>
    </postcondition>
    <target>v</target>
    <expression>v</expression>
  </assignment-proof>

  <assignment-proof ID="ass-3">
    <precondition>v</precondition>
    <postcondition>
      <normal>(v(=)v{and}(balance(=)+v))</normal>
    </postcondition>
    <target>balance</target>
    <expression>+v</expression>
  </assignment-proof>

</routine-proof>
<require>\(v \geq 0\)</require>
<ensure>balance = old(SP)balance + v</ensure>

<assignment-proof ID="ass-2">
  <precondition>\(v \geq 0\)</precondition>
  <postcondition>
    <normal>balance = old(SP)balance + v</normal>
  </postcondition>
  <target>balance</target>
  <expression>balance(v)</expression>
</assignment-proof>

</routine-proof>

<routine-proof>
  <name>withdraw</name>

  <argument>
    <name>v</name>
    <type>integer</type>
  </argument>

  <require>balance(v)</require>
  <ensure>balance = old(SP)balance - v</ensure>

  <write-attribute ID="ass-3">
    <precondition>balance(v)</precondition>
    <postcondition>
      <normal>balance = old(SP)balance - v</normal>
    </postcondition>
    <object>Current</object>
    <field>balance</field>
    <expression>balance(-v)</expression>
  </write-attribute>

</routine-proof>
5.3 Abstract Syntax Tree

To show the internal representation of the abstract syntax tree we present here the design of the tree. Each CPX_EIFFEL_PROOF consist of its attributes CPX_FEATURE and its routines CPX_ROUTINE_PROOFS. How one also can see there is for each node or XML node a pre -and postcondition node. The postcondition is of type CPX_POSTCONDITION because one can have the normal (\(Q_n\) in the theoretical part in chapter 2) or the exceptional. The exceptional states the postcondition if an error occurred (defined as \(Q_e\) in the theoretical part in chapter 2).

Figure 5.1 shows a graphical representation of the following textual syntax tree.

```
(CPX_EIFFEL_PROOF
  (CPX_CLASS_PROOF
    (STR_NAME)
    (CPX_FEATURE
      (STR_NAME)
      (STR_TYPE))
    (CPX_FEATURE
      (STR_NAME)
      (STR_TYPE))
    (CPX_ROUTINE_PROOF
      (STR_NAME)
      (CPX_ARGUMENT
        (STR_NAME)
        (STR_TYPE))
      (STR_REQUIRE
        (CONTRACT
          (NEQ
            (Exp
              (ID:n))
            (Exp
              (Void)))))
      (STR_ENSURE
        (CONTRACT
          (AND
            (EQ
              (Exp
                (ID:name))
              (Exp
                (ID:n)))
            (EQ
              (Exp
                (ID:balance))))))
  )
)
```
(Exp
  (INT:0)))))

(PIT_COMPOSITION_PROOF
  (STR_PRECONDITION
    (CONTRACT
      (NEQ
        (Exp
          (ID:n))
        (Exp
          (Void)))))
  (CPX_POSTCONDITION
    (STR_NORMAL
      (CONTRACT
        (AND
          (EQ
            (Exp
              (ID:name))
            (Exp
              (ID:n)))))
        (CPX_POSTCONDITION
          (STR_NORMAL
            (CONTRACT
              (EQ
                (Exp
                  (ID:name))
                (Exp
                  (ID:n)))))
          (STR_TARGET)
          (STR_EXPRESSION
            (CONTRACT
              (Exp

```
(ID:n)))))
(PIT_ASSIGNMENT_PROOF
(STR_PRECONDITION
(CONTRACT
(EX
(EX
(ID:name))
(EX
(ID:n))))
(CPX_POSTCONDITION
(STR_NORMAL
(CONTRACT
(AND
(EX
(ID:name))
(EX
(ID:n))
(EX
(ID:balance))
(EX
(INT:0))))
(STR_TARGET)
(STR_EXPRESSION
(CONTRACT
(EX
(INT:0))))

5.4 Eiffel Proof

The Eiffel proof representation is an readable representation of the abstract syntax tree shown before. The additional pre- and postconditions are represented in the condition clause of the Eiffel proof. This representation can be seen as a summary of the XML file. It displays the postcondition of the instruction before as well as the precondition of the instruction ahead. Otherwise the features are represented exactly the same as in the eiffel source. This representation should give the user better understanding on how the proof should look like.

The code looks as follows:

indexing
description:"Eiffel proof automatically generated by Eiffel PTC"
copyright:  "Chair of Software Engineering - ETH Zurich"
Figure 5.1: Eiffel Proof Abstract Syntax Tree
class BANK_ACCOUNT

feature
  balance: INTEGER
feature
  name: STRING
feature
  make (n: STRING) is
    -- routine generated by PTC
    require
      (n/=Void)
    do
      name := n
      condition
        normal: (name=n)
      end
      balance := 0
      condition
        normal: ((name=n) and (balance=0))
      end
    ensure
      ((name=n) and (balance=0))
    end
feature
  deposit (v: INTEGER) is
    -- routine generated by PTC
    require
      (v>0)
    do
      balance := (balance+v)
      condition
        normal: (balance=((old balance)+v))
      end
    ensure
      (balance=((old balance)+v))
    end
feature
  withdraw (v: INTEGER) is
    -- routine generated by PTC
    require
      (balance>v)
    do
      balance := (balance-v)
    end
5.5 CIL Proof

This part shows the compiled version of the CIL code and its corresponding proof. As one can see the attributes are translated into a field and the two method which sets and gets the value from the field. Noticeable is that the preconditions for an CIL instruction looks sometimes strange. For example in the withdraw feature of the bank account there is the line \((\text{balance} > v) \land (s(0) - v) = (\text{balance} - v)\) which indicates that \(\text{balance} > v \land s(0) = \text{balance}\) this strange behavior is due to the replacement function implemented in the PTC (See Section 3.1). As future work it would be nice if one can simplify this preconditions.

```java
.class public bank_account {

    .field public class INTEGER balance
    .method public INTEGER get_balance () {
        code
        
        {(balance!=Null)}                \(\text{IL001: ldld balance}\)
        
        
        
        
        {(balance!=Null) and (s(0)=balance))} \(\text{IL002: ret}\)

    }

    .method public void set_balance () {
        code
        
        {(b!=Null)}                      \(\text{IL003: ldloc b}\)
        
        
        
        
        {(b!=Null) and (s(0)=balance))} \(\text{IL004: stld balance}\)

    }

    .field public class STRING name
    .method public STRING get_name () {
        code
        
        {(name!=Null)}                   \(\text{IL001: ldld name}\)
        
        
        
        
        {(name!=Null) and (s(0)=name))} \(\text{IL002: ret}\)

    }
```
5.5. CIL Proof

.method public void set_name () {

code
{(n!=null)}
{(n!=null) and (s(0)=name)}

.method public void make (string n) {

require
(n!=null)

code
{(n!=null) and (n=n)}
{(n!=null) and (s(0)=n)}
{(name=n) and (0=0)}
{(name=n) and (s(0)=0)}

ensure
((name=n) and (balance=0))

.method public void deposit (integer v) {

require
(v>0)

code
{(v>0) and ((balance+v)=(balance+v))}
{(v>0) and ((s(0)+v)=(balance+v))}
{(v>0) and ((s(1)+s(0))=(balance+v))}

ensure
(balance=((old balance)+v))

.method public void withdraw (integer v) {

require
(balance>v)
5.5. CIL Proof

```c
code

{{(balance>v) and ((balance-v)=(balance-v))}}

{IL001: ldloc balance

{{(balance>v) and ((s(0)-v)=(balance-v))}}

{IL002: ldloc v

{{(balance>v) and ((s(1)-s(0))=(balance-v))}}

{IL003: sub

{{(balance>v) and (s(0)=(balance-v))}}

{IL004: stloc balance

ensure

(balance=((old balance)-v))

}}

```
Chapter 6

Conclusion and Future Work

To show the feasibility of the 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.

According to the transformation of the XML format into an abstract syntax tree one has gained much more flexibility. We do all the modifications and translation in the abstract syntax tree. According to this design decision it is easy to extend the abstract syntax tree, traverse it in different ways by visitors or modify the structure of the tree.

We successfully integrated a defined subset of the Eiffel language. Even the Eiffel specialities like the **and then** and the **or else** instruction we integrated into the proof-transforming compiler.

The way we did our implementation was not easy, but with the current stage one can access and modify the entire source and bytecode tree which is helpful for all the future modifications.

The new user-friendly interface allows easy navigation and highlighting of the important parts of the compilation process. It is a great help for developing and checking tasks.

As future work we plan to implement the language independent rules:

1. all-rule
2. ex-rule
3. substitution-rule
4. conjunction-rule
5. disjunction-rule

Throughout the project there are several points which can be improved or newly developed.
Simplifier

As the proofs are generated sometimes they include lines like \( s(0) = 1 \land balance = balance \) which contains no information. One should try to simplify these proofs according to some logic. For example simplify it to \( s(0) = 1 \). This can be achieved by doing one more iteration through the abstract syntax tree and apply the defined simplifications.

Integration with Isabelle

To show the feasibility of Proof-Transforming Compilers, Nordio, Karahan, Guex and this thesis [24, 11, 10] have implemented a PTC for a nearly complete subset of Eiffel. The compiler takes a proof of an Eiffel program in XML format and produces the bytecode proof. However, the bytecode proof produced as result is not embedded in any theorem prover.

A possible extension could be to embedd the Proof-Carrying Components into a theorem prover like Isabelle.

Interface Improvements

Improving the handling events in the CIL and Eiffel proofs is part of future work. It would be great if the navigation mechanism could be improved in such a way that highlights the corresponding compilation. For example on could pick an if instruction of the Eiffel proof. Then the corresponding lines of the CIL compilation should be highlighted.

The format of the CIL Proof is not optimal according to the number of parenthese used. It is possible to reduce the number of parenthesis to increase the readability of the proof. This can be achieved by going a second time over the abstract syntax tree and do the simplifications.

The pick and drop features of Eiffel should also be used for the new Interface. For example if one picks a feature in the source or in the feature tool and drops it into the Eiffel Proof Tool or CIL Tool it should go jump direct to the dropped feature.
Bibliography


## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>PTC Overview</td>
<td>4</td>
</tr>
<tr>
<td>3.1</td>
<td>Overview of PTC</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>XML node data structure</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>AST data structure used for translation</td>
<td>22</td>
</tr>
<tr>
<td>3.4</td>
<td>The Stack</td>
<td>23</td>
</tr>
<tr>
<td>3.5</td>
<td>The shift function</td>
<td>24</td>
</tr>
<tr>
<td>3.6</td>
<td>The unshift function</td>
<td>24</td>
</tr>
<tr>
<td>3.7</td>
<td>The replace function</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>Interfaces for PTC</td>
<td>30</td>
</tr>
<tr>
<td>4.2</td>
<td>The PTC default Interface</td>
<td>31</td>
</tr>
<tr>
<td>4.3</td>
<td>Header of a TOOL descriptor</td>
<td>32</td>
</tr>
<tr>
<td>4.4</td>
<td>Header of a TOOL_PANNEAL implementation</td>
<td>32</td>
</tr>
<tr>
<td>4.5</td>
<td>Registration of a tool</td>
<td>33</td>
</tr>
<tr>
<td>4.6</td>
<td>Eiffel Source Code</td>
<td>34</td>
</tr>
<tr>
<td>4.7</td>
<td>PTC Feature Tool</td>
<td>34</td>
</tr>
<tr>
<td>4.8</td>
<td>Eiffel Proof Tool</td>
<td>35</td>
</tr>
<tr>
<td>4.9</td>
<td>Highlighting conditions</td>
<td>36</td>
</tr>
<tr>
<td>4.10</td>
<td>Context Action in Eiffel Proof Tool conditions</td>
<td>37</td>
</tr>
<tr>
<td>4.11</td>
<td>CIL Tool</td>
<td>38</td>
</tr>
<tr>
<td>5.1</td>
<td>Eiffel Proof Abstract Syntax Tree</td>
<td>47</td>
</tr>
<tr>
<td>A.1</td>
<td>The PTC cluster</td>
<td>60</td>
</tr>
<tr>
<td>A.2</td>
<td>The XML parsing cluster</td>
<td>61</td>
</tr>
<tr>
<td>A.3</td>
<td>The Contract Tree cluster</td>
<td>62</td>
</tr>
<tr>
<td>A.4</td>
<td>The Proof Tree cluster</td>
<td>63</td>
</tr>
<tr>
<td>A.5</td>
<td>The Bytecode Tree cluster</td>
<td>64</td>
</tr>
<tr>
<td>B.1</td>
<td>AST_BASE</td>
<td>66</td>
</tr>
<tr>
<td>B.2</td>
<td>AST_PROOF_ITEM</td>
<td>67</td>
</tr>
<tr>
<td>B.3</td>
<td>AST_CONTENT</td>
<td>68</td>
</tr>
<tr>
<td>B.4</td>
<td>AST_VISITABLE</td>
<td>70</td>
</tr>
</tbody>
</table>
### LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.5</td>
<td>AST_EXP</td>
<td>71</td>
</tr>
<tr>
<td>B.6</td>
<td>AST_BOOLEXP</td>
<td>72</td>
</tr>
<tr>
<td>B.7</td>
<td>BC_INSTR_PROOF</td>
<td>73</td>
</tr>
<tr>
<td>B.8</td>
<td>BRANCH</td>
<td>74</td>
</tr>
<tr>
<td>B.9</td>
<td>OP</td>
<td>74</td>
</tr>
<tr>
<td>B.10</td>
<td>EB.CLICKABLE_PROOF_EDITOR</td>
<td>74</td>
</tr>
<tr>
<td>B.11</td>
<td>PTC_PANEL</td>
<td>75</td>
</tr>
</tbody>
</table>
# List of Tables

3.1 Translation of Expressions ........................................... 26
3.2 Translation of Instructions .......................................... 27
Appendix A

Clusters and Class Overview

Proof-Transforming Compiler

The proof-transforming compiler cluster (Figure A.1) is divided into several sub clusters. One for the abstract syntax tree of the bytecode, the parsing of the contracts and expressions, the abstract syntax tree of the eiffel proof and the XML parsing.

The classes in the proof-transforming compiler cluster are the implementation of the Interface described in chapter 4. The PTC class is the main starting point for each compilation.

XML Parser

These cluster (Figure A.2) is responsible for the XML parsing. See Karahan [11] for an explanation of the functions of this classes.
Eiffel Proof / Contract Tree

The contract tree (Figure A.3) represents the result of parsing a precondition, postcondition or expression. The classes starting with prefix VIS_ represent the different operators which are present in the Eiffel language. The deferred class AST_VISITABLE is the root class for the tree.

Eiffel Proof / Proof Tree

The proof tree (Figure A.4) represents the XML nodes with its rules one has to apply. Every class starting with PIT_ represents an rule. The deferred class AST_PROOF_ITEM is the root class for the tree.

Bytecode Tree

The bytecode tree (Figure A.5) presents the abstract syntax tree for the CIL code. The abstract class BC_INSTR_PROOF represents the proof for each instruction containing an proof a label and an CIL instruction. Each BC_CLASS_PROOF has a list of BC_METHODE_PROOF and this again has a list of BC_INSTR_PROOF.

In the subfolder label one can find the classes used to generate the labeling of the bytecode instructions.

Figure A.2: The XML parsing cluster
Figure A.3: The Contract Tree cluster
Figure A.4: The Proof Tree cluster
Figure A.5: The Bytecode Tree cluster
Appendix B

Class Diagrams

In this chapter I will provide an Overview of all the classes needed by the compilation Process in Bon notation.

**AST_BASE**

AST_BASE (Figure B.1) describes the topmost element of each XML node. For example the CPX_ROTINE_PROOF describes the routines with the pre-and post-condition of the whole routine and its lists of instructions.

**AST_PROOF_ITEM**

A proof item (Figure B.2) describes all the rules which can be applied. For example the PIT_ASSIGNMENT_PROOF is responsible for the translation of the assignment instruction. PIT_IF_PROOF translates the if then else instruction.

**AST_CONTENT**

AST_CONTENT (Figure B.3) is used to identify the expressions in the XML file for example in the node `< ensure > balance{=}old{SP}balance{=}v < /ensure >` the part `balance{=}old{SP}balance{=}v` is of type ECL_CONTENT. This means that it has to be parsed and translated into an abstract syntax tree of type AST_VISITABLE (Figure B.4). Others such as name are represented as simple strings.

**AST_VISITABLE**

This AST_VISITABLE (Figure B.4) type and its descendants represent the pre and postconditions as well as the expressions of Eiffel programs. As the suffix indicates
Figure B.1: AST_BASE
Figure B.2: AST_PROOF_ITEM
Figure B.3: AST_CONTENT
they are visitable with special visitors such as VIS_LINEAR or VIS_VIS_LINEAR_CIL.

**AST_EXP**

The AST_EXP (Figure B.5) represent the expressions for an AST_VISITABLE.

**AST_BOOLEXP**

The AST_BOOLEXP (Figure B.6) represent the expressions for an AST_VISITABLE (Figure B.4). They are used to describe separate the boolean operators from the other expressions.

**BC_INSTR_PROOF**

This class BC_INSTR_PROOF (Figure B.7) and its descendants represents the CIL code which consist of the precondition which is represented through the AST_VISITABLE and its descendants. The CIL Instructions and the labels are put together in BC_INSTR_PROOF (Figure B.7). The abstract classes BRANCH (Figure B.8) and OP (Figure B.9) are presented in the following. They are just there to group the instructions into several categories with the same properties.

**BRANCH**

Class BRANCH (Figure B.8) and its descendants is a subtype of BC_INSTR_PROOF. It describes the specialities of CIL branch instructions such as brtrue, bfalse and br.

**OP**

Class OP (Figure B.9) and its descendants is a subtype of BC_INSTR_PROOF. It describes the specialities of CIL operators such as add, cgl and and.

**EB_CLICKABLE_PROOF_EDITOR**

This class EB_CLICKABLE_PROOF_EDITOR (Figure B.10) describes the editor properties used in the CIL toll and the Eiffel proof tool. One can also see that the editor inherits from EB_CONTEXT_MENU_HANDLER to provide the ability of showing customized context menus.
Figure B.4: AST_VISITABLE
Figure B.5: AST_EXP
Figure B.6: AST.BOOLEXP
Figure B.7: BC_INSTR_PROOF
PTC_PANEL

PTC_PANEL (Figure B.11) is the deferred class from which every tool inherits. A tool consists of its description as well as its interface to the user the panel.