Master Thesis

Embedding proof-carrying components into Isabelle

Author(s):
Hauser, Bruno

Publication Date:
2009

Permanent Link:
https://doi.org/10.3929/ethz-a-005763266

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
Embedding Proof-Carrying Components into Isabelle

Master Thesis

By: Bruno Hauser
Supervised by: Martin Nordio
Prof. Bertrand Meyer

Student Number: 03-911-658
Abstract

The execution of mobile code can produce unexpected behaviour which may compromise security and correctness of a software system. Proof-Carrying Components are a way to overcome this problem. Proof-Carrying Components carry a mathematical proof showing that the component satisfies certain properties, known as the contract of the component. A code consumer can check the mathematical proof attached to the component before executing the mobile code. In this way, the consumer can make sure in advance that the mobile code will be executed in a safe way.

Proof-Carrying Components can be automatically generated using Proof-Transforming Compilers. Proof-Transforming Compilers are compilers that take a source proof with contracts as input and produce a bytecode proof and its contract as output. An important property of Proof-Transforming Compilers is that they do not have to be trusted. If a Proof-Transforming Compiler produces a wrong specification or a wrong proof for a component, the proof checker of the code consumer will reject the component.

In this Master thesis, we show how a bytecode proof produced as output of a Proof-Transforming Compiler can be embedded in a theorem prover. We have embedded the Proof-Carrying Components into Isabelle, using shallow embedding for the component contracts and a deep embedding for the bytecode instructions. To show that a component satisfies its contract, the generator produces a proof script. We have optimized this proof script and the measurements show that proofs are checked at least twice as fast compared to the non-optimized proof script. The embedding has been integrated into a Proof-Transforming Compiler in EiffelStudio.
Acknowledgments

I would like to thank my supervisor Martin Nordio who supported my interesting work at the Chair of Software Engineering at the Swiss Federal Institute of Technology ETH Zurich and gave many useful suggestions and ideas. My gratitude also goes to Prof. Dr. Bertrand Meyer for giving me the opportunity to write the master thesis at the Chair of Software Engineering. At last, I want to thank my parents for giving me the possibility to study at ETH Zurich and for their support throughout the years. Thank you!
Contents

1 Introduction 9

2 Formalizing the Embedding into Isabelle 13
  2.1 Background ................................................. 13
  2.1.1 Proof-Transforming Compiler ......................... 13
  2.1.2 Proof Checker .......................................... 15
  2.2 Formalization of the Embedding ....................... 16
    2.2.1 Embedding Proofs, Classes, Methods and Attributes .... 16
    2.2.2 Embedding Expressions ............................... 22
    2.2.3 Embedding Instructions .............................. 24
    2.2.4 Embedding Language Independent Rules ............... 26

3 Proof Generation Optimizations 27
  3.1 Simplifying Bytecode Proof Expressions ............. 27
    3.1.1 Simplifying EQUAL-Node Expressions .................. 28
    3.1.2 Simplifying AND-Node Expressions .................... 29
  3.2 Optimizing the Proof Script Generation ............ 30
    3.2.1 Simple Proof Script Generation ..................... 31
    3.2.2 Enhanced Proof Script Generation ................... 32
  3.3 Measurements and Comparison .......................... 37

4 Design and Implementation 39
  4.1 Overview .................................................. 39
    4.1.1 Code Generator Interface ........................... 40
    4.1.2 Code Generation Workflow ............................ 40
    4.1.3 Naming Conventions .................................. 41
  4.2 Isabelle Theory File Model ............................. 41
  4.3 Tree Visitors ............................................. 43
    4.3.1 Bytecode Proof Simplification Visitor .............. 44
    4.3.2 Skeleton Visitor ..................................... 44
    4.3.3 Contract Visitor ..................................... 44
    4.3.4 Instruction Visitor .................................. 45
  4.4 Special Generators ...................................... 46
    4.4.1 Weak and Strong Proofs ............................. 47
    4.4.2 Inheritance Set ..................................... 48
    4.4.3 Isabelle Types ..................................... 48
Chapter 1

Introduction

Today, computing architectures are becoming more and more sophisticated. Along with new modes of computing arising from the role of the internet, new modes of software deployment have emerged. Using platform independent intermediate code formats such as bytecode (.NET CIL or JVM) when distributing software is widespread nowadays. This has the advantage that it allows one to write a program in a high-level language, e.g. Eiffel [4], C# [2] or Java [6], which will then be used to generate the corresponding intermediate bytecode. But making such a software deployment process safe has become a challenge. It is dangerous to trust any entity involved. The execution of unverified mobile code can produce unexpected behaviour, which may compromise security and correctness of a software system.

 Deployed bytecode at the user-side can either be executed through interpretation or an additional compilation phase called "jitting". If one trusts the interpreter or the jitter, a way to overcome the security problem would be to apply software verification to the bytecode. However, bytecodes like .NET CIL or JVM do not have the support for high-level knowledge such as types or control flow. When generating the bytecode from the high-level source code, the high-level information cannot be carried on. For proving, this loss of information can be crucial. Apart from that, current state of the art proof development usually requires at least a limited form of interactive support from the programmer. Mostly, the development of the proof cannot be done in a fully automated fashion (for example as a byproduct of the compilation process). The fact that this interactive verification process can be eliminated if the proof delivery happens at the source code level and the loss of high-level knowledge when generating the CIL or JVM bytecode, suggests doing the correctness proofing at the source level.

 The problem that needs to be solved is how to derive from a proof at the source level a guarantee of correctness for the generated bytecode. An external entity could tamper with the bytecode, regardless whether the entity on which the compiler resides is trusted or not. Necula [23] suggested the notion of Proof-Carrying Code to solve this issue. The general idea is that before invoking the bytecode on the client side, the consumer can check an attached proof to see whether the code meets certain desirable properties. This allows the client to be sure that the execution of the bytecode will not violate any safety policy in the system and therefore not do any harm. Originally, Necula proposed in
his work to use their certifying compiler [24]. But their solution was only able to prove simple safety properties, such as type safety, automatically during the compilation process.

**Proof-Carrying Components.** Proof-Carrying Components [8] are more powerful. They are a form of trusted components for which the guarantee of quality is perhaps the strongest one possible: a mathematical proof, machine-checkable, that 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 is interesting about the component’s behavior, or just some specific aspects such as absence of “null-pointer dereferencing” or other run-time failures. Proof-Carrying Components can be automatically generated using Proof-Transforming Compilers (PTCs). PTCs are similar to certifying compilers in Proof-Carrying Code, but take a source proof as input and produce the bytecode proof. An important property of Proof-Transforming Compilers is that they do not have to be trusted. If the compiler produces a wrong specification or a wrong proof for a component, the proof checker will reject the component.

Figure 1.1 gives an overview on the Proof-Carrying Component Infrastructure. The development scheme involves the following steps:

1. On the code producer side: The source program gets verified by using proof technology at the programming level. This step can include interaction with a verification expert or the programmer.

2. On the code producer side: The source program and the proof get automatically translated into bytecode using a Proof-Transforming Compiler. This step does not need any manual interaction.
3. On the code consumer side: Before executing the bytecode through interpretation or jitting, the Isabelle code for the bytecode proof gets generated and checked by a proof checker. This is an automatic task.

Contributions. In this thesis, we use the Proof-Transforming Compiler for a subset of Eiffel developed by Nordio, Karahan, Guex, Hess and Hauser [30, 19, 17, 18]. The compiler takes a proof of an Eiffel program in XML format and produces the bytecode proof. However, the resulting bytecode proof is not yet embedded in any theorem prover in a way that its correctness can be verified by a proof checker. This work describes how Proof-Carrying Components, generated automatically by the PTC, can be embedded into Isabelle [5, 25].

The Isabelle code generator described in this thesis produces a proof script to show that a component satisfies its contract. This proof script can be optimized such that proofs are checked much faster than when using the non-optimized proof script. We describe how the enhanced proof script works in detail and present measurements that approve the proving speed gains.

Furthermore, the thesis depicts facilities to simplify bytecode proof expressions. The presented simplifier removes irrelevant subexpressions from a bytecode proof. In addition to improving the readability, this leads to slightly faster proof checking.

In order to reason about subtyping relationships, the Proof-Transforming Compiler needs to be capable of working with several classes. This thesis presents some of the extensions added to the Proof-Transforming Compiler such as the support for cluster generation and the interfacing with the Isabelle code generator.

Outline. The thesis is organized as follows: Chapter 2 gives the background to the Isabelle embedding by illustrating the Proof-Transforming Compiler and the proof checker and presents a formalization of the embedding into Isabelle. Chapter 3 presents the proof generation optimizations and measurements of the improved proof checking time. Chapter 4 is about the design and implementation of the Isabelle code generator. Chapter 5 describes the extensions applied to the Proof-Transforming Compiler. Chapter 6 presents a complete example of a bytecode proof and the corresponding generated Isabelle code. Finally, Chapter 7 concludes the thesis with a short summary and gives an outlook to possible future work.
1 Introduction
Chapter 2

Formalizing the Embedding into Isabelle

Isabelle is a generic proof assistant that allows mathematical formulas to be expressed in a formal language and provides tools for proving those formulas in a logical calculus. Its main application is the formalization of mathematical proofs and in particular formal verification, which includes proving the correctness of computer software and proving properties of computer languages and protocols.

In this thesis, we show how an Isabelle code generator can be implemented that embeds Proof-Carrying Components, produced by a Proof-Transforming Compiler, into a proof checker. The Proof-Transforming Compiler and the proof checker are two tools of the Proof-Carrying Components framework the Isabelle code generator has to work with (see Figure 1.1).

This Chapter first gives the background on the two tools by presenting a short overview of the Proof-Transforming Compiler in Section 2.1.1 and describing the properties of the proof checker in Section 2.1.2.

In the second part of this Chapter, in Section 2.2, a formalization of the embedding of Proof-Carrying Components into Isabelle is presented for bytecode proofs, classes, methods, attributes, expressions and instructions.

2.1 Background

2.1.1 Proof-Transforming Compiler

The Proof-Transforming Compiler [30, 28] is similar to a certifying compiler as presented by Necula et. al. [24], but takes a source program, its specification, and a source proof as input and produces the corresponding bytecode program, specification and proof. The Proof-Transforming Compiler consists of two modules: (1) a specification translator, transforming Eiffel contract expressions into CIL contracts based on First-Order Logic and (2) a proof translator, transforming Eiffel proofs in Hoare-style logic into CIL bytecode proofs.

Proof-Transforming Compilation is straightforward when using similar source and target languages [9, 31]. The difficulty grows with languages that have a bigger conceptual distance between the semantic models. Nordio, Müller and Meyer showed in [30] how the Proof-Transforming Compiler can transform Eif-
fel mechanisms such as multiple inheritance and contract-based exceptions to .NET CIL.

To be able to handle multiple inheritance, the Proof-Transforming Compiler maps Eiffel classes to CIL interfaces or classes, depending of the expressed source property. In most cases, the PTC will map the Eiffel classes to CIL interfaces. But in some cases, such as the ones involving reflection, the PTC transforms the Eiffel class into a CIL class instead of an interface [29]. The embedding into Isabelle pursues this approach. For simple subtype checks, the interface of a class is used and for cases involving is_equal or is_not_equal the implementation definition.

The Proof-Transforming Compilation has been formalized by Nordio et. al. and details about the PTC can be found in [29, 26, 30, 28].

Example. When feeding the Proof-Transforming Compiler with the example program in Figure 2.1 as input, the PTC produces the bytecode proof shown in Figure 2.2.

```xml
<eiffel-proof xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="../../xsd/eiffel-proof.xsd">
  <class-proof>
    <name>identity_proof</name>
    <routine-proof>
      <name>identity</name>
      <argument>
        <name>x</name>
        <type>integer</type>
      </argument>
      <require>True</require>
      <ensure>(RESULT{=}x)</ensure>
      <assignment-proof ID="assign">
        <precondition>True</precondition>
        <postcondition>
          <normal>(RESULT{=}x)</normal>
        </postcondition>
        <target>RESULT</target>
        <expression>x</expression>
      </assignment-proof>
    </routine-proof>
  </class-proof>
</eiffel-proof>
```

Figure 2.1: XML Eiffel source program input to the PTC
class public identity_proof {
    .method public void identity (integer x) {
        require
        True
        code
        {True} IL001: ldloc x
        {((True and (a(0)=x)))} IL002: stloc RESULT
        ensure
        (RESULT=x)
    }
}

Figure 2.2: Bytecode proof output from the PTC

2.1.2 Proof Checker

The proof checker used in this thesis was implemented by Nordio, Jin and Hauser [29] as a verification condition generator in Isabelle [25]. Given a bytecode proof, the verification condition generator generates a list of conditions, also called proof obligations, that have to be proven to show that the proof is correct. The proof checker proves the proof obligations by applying a given proof script. Chapter 3.2 describes how such a proof script is generated. If all proof obligations could be proven by the proof checker, the bytecode proof is correct and the component can be executed. Otherwise the proof is incorrect and the component is rejected. Figure 2.3 shows the Isabelle code of how to use the proof checker:

theorem PROGRAM_THEOREM "(is_program_safe (VCGen PROGRAM))"
PROOF_SCRIPT
done
end

Figure 2.3: Example usage of the proof checker in an Isabelle theory.

The proof checker checks the CilProgram PROGRAM by evaluating is_program_safe for the proof obligations, created by the verification condition generator VCGen for the Isabelle CilProgram PROGRAM. The theorem is then proved by applying the attached proof script PROOF_SCRIPT.

Furthermore, the proof checker defines the Isabelle types that make up an Isabelle theory file. The Isabelle theory file model described in Section 4.2 uses this type definitions as a reference for its structure. Figure 2.4 gives an overview of the CIL proof type definitions defined in the proof checker.
types Prec = "bool assn"
    Precondition = Prec
    Postcondition = "Prec × Prec"

types InstSpec = "Prec × Label × Instruction"
    CilProof = "InstSpec list"

types MethodDecl = "MethodID × Precondition × CilProof × ExcTable × Postcondition"
    ClassBody = "MethodDecl list"
    ClassDeclaration = "Type × ClassBody"
    CilProgram = "ClassDeclaration list"

Figure 2.4: Proof checker type definitions for Isabelle theory file elements.

An Isabelle theory field that is readable by the proof checker needs to consist of Isabelle constdefs elements that are of the types described in Figure 2.4. For a complete overview of all valid types, available helper functions and the control flow of the proof checker see Appendix B.1.

2.2 Formalization of the Embedding

In this Section we present the formalization of how Proof-Carrying Components can be embedded into Isabelle. We assume that the bytecode proof produced by the Proof-Transforming Compiler corresponds to the PTC output as described in [30, 29, 19, 18, 17]. Furthermore, we use the \( \nabla \) translation function to formalize the embedding of bytecode proofs from the Proof-Transforming Compiler to Isabelle. The signature of the \( \nabla \) functions looks as follows:

\[
\nabla: \text{Bytecode Proof Element} \rightarrow \text{Isabelle}
\]

where BytecodeProofElement is a CIL bytecode proof class, method or attribute and Isabelle denotes valid Isabelle code. For most of the translations this is a constant definition element, designated by constdefs, for an Isabelle type (e.g. MethodDecl) described in the proof checker in Subsection 2.1.2.

The signatures of the expression translation function \( \nabla_E \) and the instruction translation function \( \nabla_S \) are described in Subsection 2.2.2 and 2.2.3, respectively.

2.2.1 Embedding Proofs, Classes, Methods and Attributes

This Subsection formalizes the embedding of bytecode classes, methods and attributes to Isabelle. A bytecode proof consists of a list of bytecode classes with at least one element. A bytecode class can contain attributes but must consist of at least one method. The translation function \( \nabla_{Proof} \) formalizes the embedding of bytecode proofs, \( \nabla_C \) classes, \( \nabla_M \) methods and \( \nabla_A \) attributes. Further \( \nabla \) translation functions are introduced to formalize the embedding of subelements of bytecode proofs, classes, methods and attributes into Isabelle.
Variables in capital letters are used as placeholders for names (e.g. \texttt{THEORYNAME}) or Isabelle code fragments (e.g. \texttt{THEOREM_PROOF}).

\textbf{Bytecode Proof}

Let $p$ be a bytecode proof defined as $p = c_1...c_n$, where $c_i$ is a bytecode class proof and $1 \leq i \leq n$. The translation $\nabla_{\text{Proof}}$ is defined as follows:

\[
\nabla_{\text{Proof}}, \nabla_{\text{IsabelleHeader}}, \nabla_{\text{IsabelleTail}}: \text{Bytecode Proof} \rightarrow \text{Isabelle}
\]

$\nabla_{\text{Proof}}(p) = \nabla_{\text{IsabelleHeader}}(p) + \nabla_C(c_1) + ... + \nabla_C(c_n) + \nabla_{\text{IsabelleTail}}(p)$

The $\nabla_{\text{IsabelleHeader}}$ and $\nabla_{\text{IsabelleTail}}$ are defined as:

\[
\nabla_{\text{IsabelleHeader}}(p) = \nabla_{\text{Preamble}}(p) + \nabla_{\text{Types/IDs}}(p)
\]

\[
\nabla_{\text{IsabelleTail}}(p) = \nabla_{\text{Program}}(p) + \nabla_{\text{SubtypeSet}}(p) + \nabla_{\text{TheoremProof}}(p)
\]

In the following, we present the definitions of the translation functions that make up the $\nabla_{\text{IsabelleHeader}}$ and the $\nabla_{\text{IsabelleTail}}$ translation function. The signature of these translation functions is:

\[
\nabla_{\text{Preamble}}, \nabla_{\text{Types/IDs}}, \nabla_{\text{Program}}, \nabla_{\text{SubtypeSet}}, \nabla_{\text{TheoremProof}}: \text{Bytecode Proof} \rightarrow \text{Isabelle}
\]

The $\nabla_{\text{Preamble}}$ translation is defined as follows, where \texttt{THEORYNAME} must be equivalent to the Isabelle theory filename without the .thy ending and \texttt{PROOFCHECKER} denotes the name of the proof checker theory file without the .thy ending:

\[
\begin{align*}
\text{theory} & \quad \text{THEORYNAME} \\
\text{imports} & \quad \text{PROOFCHECKER} \\
\text{begin} & \quad \text{end}
\end{align*}
\]

The $\nabla_{\text{Types/IDs}}$ translation function embeds the bytecode class interfaces with their implementations and all method IDs of all classes $c_i$ of the bytecode proof to Isabelle. \texttt{CLASSNAME} denotes the name of the class $c_i$, \texttt{METHODNAME} the name of the method $m_j$ in class $c_i$ and \texttt{METHODNUMBER} a unique integer ID for the method $m_j$.

\[
\begin{align*}
\text{consts} & \quad \text{CLASSNAME_type:: TName} \\
& \quad \text{CLASSNAME_INTERFACE_type:: TName}
\end{align*}
\]

\[
\begin{align*}
\text{constdefs} & \quad \text{METHODNAMEID:: MethodID} \\
& \quad \text{"METHODNAMEID} \equiv \text{METHODNUMBER}"
\end{align*}
\]

The $\nabla_{\text{Program}}$ function translates to the Isabelle code below:

\[
\begin{align*}
\text{constdefs} & \quad \text{PROGRAM:: CilProgram} \\
& \quad \text{[simp]: "PROGRAM} \equiv \text{[CLASSNAME_1,...,CLASSNAME_N]"
\end{align*}
\]

where \texttt{CLASSNAME_i} in \texttt{CLASSNAME_1,...,CLASSNAME_N} corresponds to the name of class $c_i$ of the bytecode proof. The meaning of \texttt{PROGRAM} is described in the $\nabla_{\text{TheoremProof}}$ translation function.
The $\nabla_{\text{SubtypeSet}}$ is defined as follows:

\[
defs \text{cil\_subtypeSet\_def[simp]}: \text{cil\_subtypeSet} \equiv
\{(\text{TPNAME}_x, \text{TPNAME}_y) \ldots (\text{TPNAME}_{xm}, \text{TPNAME}_{yn})\}
\]

where a tuple $(\text{TPNAME}_x, \text{TPNAME}_y)$ denotes a subtyping relationship for the bytecode classes with the typenames $\text{TPNAME}_x$ and $\text{TPNAME}_y$, respectively. The typenames $\text{TName}$ for bytecode class interfaces and implementations are defined by $\nabla_{\text{Types/IDs}}$.

The $\nabla_{\text{TheoremProof}}$ function translates to the following Isabelle code:

\[
\text{theorem PROGRAM\_THEOREM } "(\text{is\_program\_safe } \text{VCGen PROGRAM})"\\
\quad \text{THEOREM\_PROOF}\\
\quad \text{done}\\
\quad \text{end}
\]

$\text{PROGRAM\_THEOREM}$ is the name of the theorem that proofs the Isabelle code and $\text{PROGRAM}$ corresponds to the name of the program. These two variables do not have a direct counterpart in the bytecode proof and can therefore be chosen freely. The $\text{THEOREM\_PROOF}$ denotes the generated proof script for the theorem $\text{PROGRAM\_THEOREM}$. The Algorithm 3.3 in Chapter 3.2.2 describes the Isabelle code that will replace the $\text{THEOREM\_PROOF}$ variable.

**Bytecode Classes**

Let $c_i$ be a bytecode class defined as $c_i = [a_1...a_k] + r_1...r_m$, where $a_h$ is a bytecode attribute, $r_j$ is a bytecode routine proof and $1 \leq h \leq k$ as well as $1 \leq j \leq m$. Since bytecode classes do not necessarily have attributes ($k$ can be 0), it is also valid if no attributes are embedded into Isabelle.

$\nabla_C, \nabla_{\text{ClassBody}}, \nabla_{\text{ClassDecl}}: \text{Bytecode Class Proof} \rightarrow \text{Isabelle}$

\[
\nabla_C(c_i) = [\nabla_A(a_1) + \ldots + \nabla_A(a_k)] \nabla_M(r_1) + \ldots + \nabla_M(r_m) + \nabla_{\text{ClassBody}}(c_i) + \nabla_{\text{ClassDecl}}(c_i)
\]

$\nabla_{\text{ClassBody}}$ is defined as:

\[
\text{constsdefs}\\
\quad \text{CLASSNAME}_i\_body :: \text{ClassBody}\\
\quad \text{[simp]}:\"\text{CLASSNAME}_i\_body \equiv \text{[METHODNAME}_1,\ldots,\text{METHODNAME}_M]\"\]

And the $\nabla_{\text{ClassDecl}}$ translation function yields:

\[
\text{constsdefs}\\
\quad \text{CLASSNAME}_i :: \text{ClassDeclaration}\\
\quad \text{[simp]}:\"\text{CLASSNAME}_i \equiv \text{[refT CLASSNAME}_i\_type, \text{CLASSNAME}_i\_body]\"\]

where $\text{CLASSNAME}_i$ corresponds to the name of the bytecode class $c_i$. For $\text{CLASSNAME}_i\_body$ the same holds, except that in addition _body is appended to the classname. $\text{METHODNAME}_j$ is equal to the name of the bytecode routine $r_j$. 
Formalizing the Embedding into Isabelle

2 Bytecode Methods

Let \( r_j \) be a bytecode method defined as

\[
 r_j = \text{prec} + \text{post} + \text{arg} + [l_1...l_p] + \text{inst\_proof\_list}.
\]

Since the Proof-Transforming Compiler assumes that each bytecode method always has an argument [18], we use \( \text{arg} \) to denote this element. The \( l_r \) in \([l_1...l_p]\) denotes a possible local variable, for which \( 1 \leq r \leq p \). \( \text{inst\_proof\_list} \) corresponds to the list of bytecode proof instructions making up the bytecode method.

\[
\nabla_M: \text{Bytecode Method Proof} \rightarrow \text{Isabelle}
\]

\[
\nabla_M(r_j) = \nabla_{\text{Prec}}(r_j) + \nabla_{\text{Post}}(r_j) + \nabla_{\text{Local}}(\text{arg}) + [\nabla_{\text{Local}}(l_1) + ... + \nabla_{\text{Local}}(l_p)] + \nabla_{\text{Inst\_Proof\_List}}(r_j) + \nabla_{\text{Method\_Body}}(r_j) + \nabla_{\text{Exception}}(r_j)
\]

We use \( \text{prec} \) and \( \text{post} \) to denote the bytecode proof methods precondition and the postcondition, respectively. \( \text{prec} \) and \( \text{post} \) are both of type \( \text{EiffelContract} \). The definition of the \( \text{EiffelContract} \) type is described in Nordio [29]. For a detailed description see [29].

The signature for the \( \nabla_{\text{Prec}}, \nabla_{\text{Post}} \) and \( \nabla_{\text{Local}} \) translation functions is the following:

\[
\nabla_{\text{Prec}}, \nabla_{\text{Post}}, \nabla_{\text{Local}}: \text{Bytecode Method Proof} \rightarrow \text{Isabelle}
\]

The \( \nabla_{\text{Prec}} \) translation function is defined as follows, where the \( \text{METHODNAME}_j \) designates the name of the corresponding bytecode method \( r_j \).

\[
\text{constdefs}
\]

\[
\text{METHODNAME}_j\text{PreCond}:: \text{Prec}
\equiv \lambda s \sigma Z. E(\text{prec})
\]

and \( \nabla_{\text{Post}} \) as:

\[
\text{constdefs}
\]

\[
\text{METHODNAME}_j\text{PostCond}:: \text{Prec}
\equiv \lambda s \sigma Z. E(post)
\]

\[
\text{constdefs}
\]

\[
\text{METHODNAME}_j\text{PostCondE}:: \text{Prec}
\equiv \lambda s \sigma Z. False
\]

The expression translation function \( \nabla_E \) is described in Section 2.2.2.

The \( \nabla_{\text{Local}} \) function translates to the following Isabelle code:

\[
\text{constdefs}
\]

\[
\text{METHODNAME}_j\_\text{VARIABLENAME}_r:: \text{VarName}
\equiv \text{VARIABLENUMBER}
\]

where the \( \text{VARIABLENAME}_r \) denotes the name of the argument variable \( \text{arg} \) or a local variable \( l_r \) respectively. \( \text{VARIABLENUMBER} \) is a unique integer ID for the local \( l_r \) and the argument \( \text{arg} \).
The $\nabla_{\text{InstProofList}}$ translation function is defined as:

$$
\nabla_{\text{InstProofList}}: \text{Bytecode Method Proof} \rightarrow \text{List}[\text{Bytecode Instruction Proof}]
$$

$$
\nabla_{\text{InstProofList}}(\text{inst\_proof\_list}) = \nabla_{\text{InstProof}}(\text{inst\_proof\_1}) + ... + \nabla_{\text{InstProof}}(\text{inst\_proof\_w})
$$

and the $\nabla_{\text{InstProof}}$, $\nabla_{\text{InstSpec}}$ and $\nabla_{\text{InstPrec}}$ functions as:

$$
\nabla_{\text{InstProof}}: \text{Bytecode Instruction Proof} \rightarrow \text{Isabelle}
$$

$$
\nabla_{\text{InstSpec}}: \text{Bytecode Instruction Proof} \rightarrow \text{Bytecode Instruction}
$$

$$
\nabla_{\text{InstPrec}}: \text{Bytecode Instruction Proof} \rightarrow \text{EiffelContract}
$$

The translation function $\nabla_{\text{InstSpec}}$ yields the bytecode instruction and $\nabla_{\text{InstPrec}}$ the bytecode instructions precondition. For a function $\nabla_{\text{InstProof}}(\text{inst\_proof\_v})$ in $\nabla_{\text{InstProof}}(\text{inst\_proof\_1}) + ... + \nabla_{\text{InstProof}}(\text{inst\_proof\_w})$ the following holds: $1 \leq v \leq w$.

The $\nabla_{\text{InstProof}}$ function translates to the Isabelle code below:

\begin{verbatim}
constdefs
  P[NUMBER]METHODNAME jBODY:: Prec
  "P[NUMBER]METHODNAME jBODY ≡ (λ s σ Z. \nabla E (\nabla InstPrec (inst\_proof\_v)))"

constdefs
  L[NUMBER]METHODNAME jBODY:: InstSpec
  "L[NUMBER]METHODNAME jBODY ≡ (P[NUMBER]METHODNAME jBODY,
  [NUMBER], \nabla S (\nabla InstSpec (inst\_proof\_v)))"
\end{verbatim}

where $\text{NUMBER}$ designates a unique integer ID for the instruction proof ($\text{inst\_proof\_v}$). The expression translation function $\nabla E$ is described in Subsection 2.2.2 and the instruction translation function $\nabla S$ in Subsection 2.2.3.

The signature for the three translation functions $\nabla_{\text{MethodBody}}$, $\nabla_{\text{Exception}}$ and $\nabla_{\text{MethodDecl}}$ is:

$$
\nabla_{\text{MethodBody}}, \nabla_{\text{Exception}}, \nabla_{\text{MethodDecl}}:
\text{Bytecode Method Proof} \rightarrow \text{Isabelle}
$$

The $\nabla_{\text{MethodBody}}$ function translates to the Isabelle code below:

\begin{verbatim}
constdefs
  METHODNAME jBODY:: CilProof
  "METHODNAME jBODY ≡ [INSTRUCTIONNAME_1,...,INSTRUCTIONNAME_w]"
\end{verbatim}

$\text{INSTRUCTIONNAME}_1,...,\text{INSTRUCTIONNAME}_w$ denote the names of the translated bytecode instruction Isabelle elements, e.g. $\text{INSTRUCTIONNAME}_1$, when applying function $\nabla S$ to the bytecode method body instructions from $\text{instructions}$. 
The $\nabla_{\text{Exception}}$ is defined as:
\[
\text{constdefs}
\quad \text{METHODNAME}\_\text{jet} \:: \text{ExcTable}
\quad \text{"METHODNAME}\_\text{jet} \equiv \{\nabla_{\text{ETentry}}\}\]
\]
where $\nabla_{\text{ETentry}}$ denotes the embedding of exception table entries to Isabelle.
The $\nabla_{\text{ETentry}}$ translation function is described in Section 2.2.3 and is used when
the function $\nabla_S$ is applied to a try-catch bytecode instruction.

The $\nabla_{\text{MethodDecl}}$ translation function is defined as:
\[
\text{constdefs}
\quad \text{METHODNAME}\_j \:: \text{MethodDecl}
\quad \text{[simp]:}"\text{METHODNAME}\_j \equiv
\quad \text{(METHODNAME}\_\text{jID}, \text{METHODNAME}\_\text{jPreCond}, \text{METHODNAME}\_\text{jBODY},
\quad \text{METHODNAME}\_\text{jet}, \text{(METHODNAME}\_\text{jPostCond}, \text{METHODNAME}\_\text{jPostCondE})}\"
\]
where the $\text{METHODNAME}\_j$ and $\text{METHODNAME}\_j^*$ name variables correspond to the
name variables described in the translation functions above (e.g. $\text{METHODNAME}\_\text{jID}$
from $\nabla_{\text{Types/IDs}}$).

**Bytecode Attributes**

Let $a_h$ be a bytecode attribute. In the following definitions, $\text{FIELDNAME}A$ designates
the name of the bytecode attribute $a_h$. The translation $\nabla_A$ is defined as follows:
\[
\nabla_A \:: \text{Bytecode Field} \rightarrow \text{Isabelle}
\quad \nabla_A(a_h) = \nabla_{\text{FieldType}}(a_h) + \nabla_{\text{FieldDecl}}(a_h) + (a_h)
\]

The $\nabla_{\text{FieldType}}$ is defined as:
\[
\text{constdefs}
\quad \text{FIELDNAME}h\_\text{type} \:: \text{Type}
\quad \text{"FIELDNAME}h\_\text{type} \equiv \text{TYPE}\"
\]
where TYPE corresponds to a built-in Isabelle type $\text{intT}$ or $\text{boolT}$ or to a type
constructed with typename (refT $\text{TPNAME}$). $\text{TPNAME}$ denotes a name for a
typename of type $\text{TName}$ for bytecode class interfaces and implementations as
defined by $\nabla_{\text{Types/IDs}}$.

The $\nabla_{\text{FieldDecl}}$ function translates to the Isabelle code below:
\[
\nabla_{\text{FieldDecl}} \:: \text{Bytecode Field} \rightarrow \text{Isabelle}
\quad \text{constdefs}
\quad \text{FIELDNAME}h\_\text{decl} \:: \text{FDecl}
\quad \text{"FIELDNAME}h\_\text{decl} \equiv \text{(FIELDNUMBER, FIELDNAME}h\_\text{type)}"
\]
where $\text{FIELDNUMBER}$ denotes a unique integer ID for the attribute $a_h$. 
The $\nabla_{\text{Field}}$ function translates to the following Isabelle code:

$$
\begin{align*}
\text{constdefs} \\
\text{FIELDNAME}_h :: \text{IFldId} \\
\text{"FIELDNAME}_h \equiv (\text{FIELDNAME}_h\text{\_dec}, \text{FIELDTYPENAME})"
\end{align*}
$$

where $\text{FIELDTYPENAME}$ denotes for the typename of type $\text{TName}$ belonging to the corresponding $\text{TYPE \ FIELDNAME}_h\text{\_type}$, as described by the $\nabla_{\text{FieldType}}$ translation function above.

### 2.2.2 Embedding Expressions

In this Subsection, we present the definition of $\nabla_{E}$, the translation function for the embedding of contract expressions. The signature of the expression translation function is specified as:

$$
\nabla_{E} : \text{EiffelContract} \rightarrow \text{Isabelle}
$$

To provide an indication, we informally describe the elements that constitute an $\text{EiffelContract}$. $\text{EiffelContracts}$ are expressed using boolean expressions. Boolean expressions are logical operators compositions of expressions and type functions. Expressions are constants, local variables and parameters, attributes, routine calls, creation expressions, old expressions, boolean expressions or void. The type functions are $\text{ConformsTo}$, $\text{IsEqual}$ or $\text{IsNotEqual}$. The $\text{EiffelContract}$ type is specified in detail by Nordio [29]. We refer to the definitions of the $\text{EiffelContract}$ type there and use the $\nabla_{b}$ translation function for boolean expressions and the $\nabla_{\text{exp}}$ function for expressions. The $\nabla_{E}$ translation function can be written as:

$$
\nabla_{E} : \nabla_{b} \text{ or } \nabla_{\text{exp}} \rightarrow \text{Isabelle}
$$

In the following, we give the specification of other translations used to formalize the embedding of contract expressions:

Function $\nabla_{\text{type.eq}}$ yields the Isabelle type $\text{aInt}$, $\text{aBool}$, $\text{intV}$ or $\text{boolV}$ as defined by Algorithm 4.2 in Section 4.4.3 and function $\nabla_{\text{type.binop}}$ returns the Isabelle type according to Algorithm 4.1 in Section 4.4.3.

The function $\nabla_{\text{IsabelleType}}$, given an Eiffel class type expression, returns the respective Isabelle type:

$$
\begin{align*}
\nabla_{\text{IsabelleType}} : \text{EType} \rightarrow \text{Isabelle} \\
\nabla_{\text{IsabelleType}}(\text{INTEGER}) &= \text{intT} \\
\nabla_{\text{IsabelleType}}(\text{BOOLEAN}) &= \text{boolT} \\
\nabla_{\text{IsabelleType}}(\text{EType}) &= \text{refT \ CLASSNAME\_type}
\end{align*}
$$

where $\text{CLASSNAME}$ denotes the name of the bytecode/eiffel class and $\text{CLASSNAME\_type}$ is of type $\text{TName}$ as described by the translation function $\nabla_{\text{Types/IDs}}$. 
The definition of the functions $\nabla_b$ and $\nabla_{exp}$ is the following:

\[
\begin{align*}
\nabla_b(\text{Eq } e_1 e_2) &= \nabla_{\text{type,eq}}(\nabla_{\text{exp}}(e_1)) = \nabla_{\text{type,eq}}(\nabla_{\text{exp}}(e_2)) \\
\nabla_b(\text{NotEq } e_1 e_2) &= \nabla_{\text{type,eq}}(\nabla_{\text{exp}}(e_1)) \sim \nabla_{\text{type,eq}}(\nabla_{\text{exp}}(e_2)) \\
\nabla_b(\text{Impl } b_1 b_2) &= \nabla_{\text{type,binop}}(\nabla_b(b_1)) \rightarrow \nabla_{\text{type,binop}}(\nabla_b(b_2)) \\
\nabla_b(\text{Xor } b_1 b_2) &= (\nabla_{\text{type,binop}}(\nabla_b(b_1)) \land \nabla_{\text{type,binop}}(\nabla_b(b_2))) \lor (\nabla_{\text{type,binop}}(\nabla_b(b_1)) \land \neg \nabla_{\text{type,binop}}(\nabla_{\text{exp}}(b_2))) \\
\nabla_b(\text{Or } b_1 b_2) &= \nabla_{\text{type,binop}}(\nabla_b(b_1)) \lor \nabla_{\text{type,binop}}(\nabla_b(b_2)) \\
\nabla_b(\text{OrElse } b_1 b_2) &= \text{if } \nabla_{\text{type,binop}}(\nabla_b(b_1)) \text{ then True else } \nabla_{\text{type,binop}}(\nabla_b(b_2)) \\
\nabla_b(\text{And } b_1 b_2) &= (\nabla_{\text{type,binop}}(\nabla_b(b_1)) \land \nabla_{\text{type,binop}}(\nabla_b(b_2))) \\
\nabla_b(\text{AndThen } b_1 b_2) &= \text{if } \neg \nabla_{\text{type,binop}}(\nabla_b(b_1)) \text{ then False else } \nabla_{\text{type,binop}}(\nabla_b(b_2)) \\
\nabla_b(\text{Neg } b) &= \neg \nabla_{\text{type,binop}}(\nabla_b(b)) \\
\nabla_b(\text{True}) &= \text{True} \\
\nabla_b(\text{False}) &= \text{False} \\
\nabla_b(\text{Less } e_1 e_2) &= \text{aInt } \nabla_{\text{exp}}(e_1) < \text{aInt } \nabla_{\text{exp}}(e_2) \\
\nabla_b(\text{LessE } e_1 e_2) &= \text{aInt } \nabla_{\text{exp}}(e_1) <= \text{aInt } \nabla_{\text{exp}}(e_2) \\
\nabla_b(\text{Greater } e_1 e_2) &= \text{aInt } \nabla_{\text{exp}}(e_1) > \text{aInt } \nabla_{\text{exp}}(e_2) \\
\nabla_b(\text{GreaterE } e_1 e_2) &= \text{aInt } \nabla_{\text{exp}}(e_1) >= \text{aInt } \nabla_{\text{exp}}(e_2) \\
\n\nabla_{\text{exp}}(\text{ConformsTo } t_1 t_2) &= \text{subtype } (\nabla_{\text{IsabelleType}}(t_1) \nabla_{\text{IsabelleType}}(t_2)) \\
\n\nabla_{\text{exp}}(\text{IsEqual } t_1 t_2) &= \text{subtype } (\nabla_{\text{IsabelleType}}(t_1) \nabla_{\text{IsabelleType}}(t_2)) \land \text{subtype } (\nabla_{\text{IsabelleType}}(t_2) \nabla_{\text{IsabelleType}}(t_1)) \\
\n\nabla_{\text{exp}}(\text{IsNotEqual } t_1 t_2) &= \neg (\text{subtype } (\nabla_{\text{IsabelleType}}(t_1) \nabla_{\text{IsabelleType}}(t_2)) \land \text{subtype } (\nabla_{\text{IsabelleType}}(t_2) \nabla_{\text{IsabelleType}}(t_1))) \\
\n\nabla_{\text{exp}}(\text{Void}) &= \text{nullV} \\
\n\nabla_{\text{exp}}(\text{Int } i) &= \text{intV } i \\
\n\nabla_{\text{exp}}(\text{ID } \text{var}) &= (\text{getV } \sigma \text{ var})
\[ \nabla_{\text{exp}}(\text{Nil}) = \text{nullV} \]

\[ \nabla_{\text{exp}}(\text{Stack } x) = (s \mid x) \]

\[ \nabla_{\text{exp}}(\text{Plus } e_1 e_2) = \text{intV } ((\text{aInt } \nabla_{\text{exp}}(e_1)) + (\text{aInt } \nabla_{\text{exp}}(e_2))) \]

\[ \nabla_{\text{exp}}(\text{Minus } e_1 e_2) = \text{intV } ((\text{aInt } \nabla_{\text{exp}}(e_1)) - (\text{aInt } \nabla_{\text{exp}}(e_2))) \]

\[ \nabla_{\text{exp}}(\text{Mul } e_1 e_2) = \text{intV } ((\text{aInt } \nabla_{\text{exp}}(e_1)) \times (\text{aInt } \nabla_{\text{exp}}(e_2))) \]

\[ \nabla_{\text{exp}}(\text{Div } e_1 e_2) = \text{intV } ((\text{aInt } \nabla_{\text{exp}}(e_1)) \div (\text{aInt } \nabla_{\text{exp}}(e_2))) \]

\[ \nabla_{\text{exp}}(\text{New store type}) = \text{refV } (\text{newIns CLASSNAME_type } (\text{store})) \]

\[ \nabla_{\text{exp}}(\text{Alloc store type}) = \text{iAlloc store CLASSNAME_type} \]

\[ \nabla_{\text{exp}}(\text{Access store type attribute}) = \text{readOS store } (\text{iLoc(\text{valO(getV } \sigma \text{ CLASSNAME})) attribute}) \]

\[ \nabla_{\text{exp}}(\text{Update store type attribute}) = \text{updtOS store } (\text{iLoc(\text{valO(getV } \sigma \text{ CLASSNAME})) attribute}) \]

where \text{store} designates the abstract data type for an object store. We use the object store presented in [32]. The signatures for the \text{New}, \text{Access}, \text{Alloc} and \text{Update} object store operations can be found in [29].

### 2.2.3 Embedding Instructions

In this Subsection, we present the definition of \( \nabla_S \), the translation function for the embedding of instructions. The signature of the instruction translation function looks as follows:

\[ \nabla_S : \text{Bytecode Instruction} \rightarrow \text{Isabelle} \]

\[ \nabla_S(\text{brline}) = \text{br line} \]

\[ \nabla_S(\text{brfalse line}) = \text{brfalse line} \]

\[ \nabla_S(\text{brtrue line}) = \text{brtrue line} \]

\[ \nabla_S(\text{leaveline}) = \text{br line} \]

\[ \nabla_S(\text{callvirt object method param}) = \text{callvirt CLASSNAME_type methodID param} \]

\[ \nabla_S(\text{ldc}) = \text{ldc } \nabla_{\text{IsabelleValueTypes}} \]

\[ \nabla_S(\text{ldfld}) = \text{ldfld} \]

\[ \nabla_S(\text{ldloc}) = \text{ldloc} \]
\( \nabla_S(\text{ret}) = \text{ret} \)
\( \nabla_S(\text{stloc}) = \text{stloc} \)
\( \nabla_S(\text{stfld}) = \text{stfld} \)
\( \nabla_S(\text{throw}) = \text{throw} \)
\( \nabla_S(\text{rethrow}) = \text{throw} \)
\( \nabla_S(\text{newobjobject}) = \text{newobj CLASSNAME} \)
\( \nabla_S(\text{add}) = \text{iladd} \)
\( \nabla_S(\text{sub}) = \text{ilsub} \)
\( \nabla_S(\text{and}) = \text{iland} \)
\( \nabla_S(\text{or}) = \text{ilor} \)
\( \nabla_S(\text{ceq}) = \text{ilceq} \)
\( \nabla_S(\text{cgt}) = \text{ilcgt} \)
\( \nabla_S(\text{cgte}) = \text{ilcgte} \)
\( \nabla_S(\text{clt}) = \text{ilclt} \)
\( \nabla_S(\text{clte}) = \text{ilclt} \)
\( \nabla_S(\text{cneq}) = \text{ilcneq} \)
\( \nabla_S(\text{div}) = \text{ildiv} \)
\( \nabla_S(\text{mul}) = \text{ilmul} \)
\( \nabla_S(\text{nop}) = \nabla_{\text{WeakStrong}}(\text{nop}) \)
\( \nabla_S(\text{try} - \text{catch}) = \nabla_{\text{ETentry}}(\text{try} - \text{catch}) \)

where \texttt{CLASSNAME} \_\texttt{type} designates the corresponding class of \texttt{object} and \( \nabla_{\text{IsabelleValue}} \)\texttt{Type} yields the Isabelle value type \texttt{boolV}, \texttt{intV} or \texttt{nullV} depending on the type of the constant to load. The \( \nabla_{\text{WeakStrong}} \) function is described in Subsection 2.2.4. The \( \nabla_{\text{ETentry}} \) translation function creates an Isabelle exception table entry for the exception table specified in Subsection 2.2.1. The \( \nabla_{\text{ETentry}} \) is defined as:

\[
\nabla_{\text{ETentry}} : \text{Bytecode Try Catch} \rightarrow \text{Isabelle}
\]

and applying \( \nabla_{\text{ETentry}} \) to a \texttt{try} \_\texttt{catch} bytecode instruction proof embeds the following Isabelle code below:
(try − start, try − end, catch − start, (refT EXCEPTION type))

The try − start denotes the number of the Isabelle instruction specification element where the try block starts, try − end where the try block ends and catch − start where the catch block starts. This tells the proof checker where to jump in case e.g. a throw instruction was detected in the try block.

2.2.4 Embedding Language Independent Rules

In this Subsection, we present the translation of the language-independent strong and weak rules. The signature of the language independent rules translation function looks as follows:

\[ \nabla_{\text{WeakStrong}}: \text{Bytecode Nop} \rightarrow \text{Isabelle} \]

As defined by Nordio in [29], the nop bytecode instruction can contain a proof for a weak \((Q_n \Rightarrow Q'_n)\) or a strong \((P' \Rightarrow P)\) rule translation. The following Isabelle code is translated for \(\nabla_{\text{WeakStrong}}(\text{nop})\):

\[ \text{lemma LEMMANAME: "LEFT IMPPLICATION BOOLEXP --> RIGHT IMPlication BOOLEXP"} \]

LEMMANAME designates the name of the lemma, generated from the name of the method containing the weak/strong proof and a unique ID. LEFT IMPlication BOOLEXP stands for the \(\nabla_E(Q_n)\) and RIGHT IMPlication BOOLEXP for the \(\nabla_E(Q'_n)\) translated Isabelle expression, respectively.
Chapter 3

Proof Generation
Optimizations

Proof-Carrying Components are embedded into Isabelle using the translation formalizations presented in Chapter 2. To show that a component satisfies its contract, the Isabelle code generator produces a proof script. This chapter describes how the bytecode proof of a Proof-Carrying Component can be simplified to be more readable before it gets embedded into Isabelle and how the proof script generation can be optimized. Figure 3.1 shows an overview of the optimizations developed in this thesis.

The first optimization, A in Figure 3.1, is applied directly on the bytecode proof produced by the Proof-Transforming Compiler. Expressions of the bytecode proof are optimized in such a way that subexpressions holding no useful information are eliminated. Section 3.1 explains this step in more detail.

The second optimization, B in Figure 3.1, is applied during the Isabelle code generation. The Isabelle code generator, described in more detail in Chapter 4, takes a bytecode proof as input and generates the corresponding Isabelle code as output. However, there are different ways how the proof for the Isabelle code output can be generated. Section 3.2 describes a simple and a tweaked way of generating the Isabelle theorem proof. Section 3.3 concludes with a comparison of their proving speed for several Isabelle theory files.

3.1 Simplifying Bytecode Proof Expressions

When working with proof-annotated bytecode consisting of complex preconditions, the bytecode proof produced by the Proof-Transforming Compiler can get quite long and confusing. The Proof-Transforming Compiler translates expressions using the weakest precondition [14]. For example, applying the weakest precondition to the assignment \( x := e \) with its postcondition \( \{ P \} \) yields predicate \( \{ P[e/x] \} \). This replacement produces expressions such as \( \{ \text{Result} := x + 1 \land y + 5 = y + 5 \} \). Since the subexpression \( y + 5 = y + 5 \) simplifies to true, the whole expression can be reduced to \( \{ \text{Result} := x + 1 \} \). When applied systematically, such simplifications will avoid blown-up bytecode proof expressions, improve the readability of the proof and also optimize the proof checking, since less expressions have to be verified.
We have implemented a bytecode proof expression simplifier using the visitor pattern [15]. The simplifications are employed directly on the bytecode proof coming from the Proof-Transforming Compiler. Step A in Figure 3.1 designates, where the optimizations are applied in the Proof-Carrying Components framework. In the following, we present two examples of such simplifications.

### 3.1.1 Simplifying EQUAL-Node Expressions

The computing of long arithmetic expressions can lead to long instruction preconditions. The example in Figure 3.2 illustrates how the application of the weakest precondition can blow up the preconditions of the instructions when computing $balance := balance + v$. Since such weakest precondition replacements yield to inexpressive subexpressions, not containing any important information regarding the correctness of the proof, they can be simplified. Figure 3.3 shows the same bytecode example after the simplifications described by the pseudocode algorithm in 3.1 were applied.

\[
\begin{align*}
\{ v > 0 \land (balance + v) = (balance + v) \} & \quad IL001 : ldloc balance \\
\{ v > 0 \land (s(0) + v) = (balance + v) \} & \quad IL002 : ldloc v \\
\{ v > 0 \land (s(1) + s(0)) = (balance + v) \} & \quad IL003 : add \\
\{ v > 0 \land (s(0)) = (balance + v) \} & \quad IL004 : stloc balance
\end{align*}
\]

Figure 3.2: Bytecode proof example for arithmetic expressions
\{v > 0\} \quad IL001 : ldloc balance
{v > 0 \land (s(0)) = (balance)} \quad IL002 : ldloc v
{v > 0 \land (s(1) + s(0)) = (balance + v)} \quad IL003 : add
{v > 0 \land (s(0)) = (balance + v)} \quad IL004 : stloc balance

Figure 3.3: Simplified bytecode proof for the example in 3.2

Algorithm 3.1 Simplification algorithm to eliminate binary operation subexpressions in EQUAL-Nodes as in the example 3.3

\textbf{if} \quad \text{EQ\_NODE.left.type} \preceq \text{BINARY\_OP} \quad \land \quad \text{EQ\_NODE.right.type} \preceq \text{BINARY\_OP} \quad \land \quad \text{EQ\_NODE.left.type} = \text{EQ\_NODE.right.type} \quad \textbf{then}
\text{EQ\_NODE.left} \leftarrow \text{EQ\_NODE.left.right}
\text{EQ\_NODE.right} \leftarrow \text{EQ\_NODE.right.right}
\textbf{else}
\text{EQ\_NODE.left.right} = \text{EQ\_NODE.right.right} \quad \textbf{then}
\text{EQ\_NODE.left} \leftarrow \text{EQ\_NODE.left.left}
\text{EQ\_NODE.right} \leftarrow \text{EQ\_NODE.right.left}
\textbf{end if}
\textbf{end if}

3.1.2 Simplifying AND-Node Expressions

Applying the weakest precondition to variable assignments yields to unnecessarily long instruction preconditions. The bytecode proof in Figure 3.4 illustrates an example. Since the subexpressions $n = n$ and $0 = 0$ are tautologies, they are simply replaced by $true$. This results in a cleaner and less confusing bytecode proof as shown in Figure 3.5. The algorithm in 3.2 presents the pseudocode for the described simplification.

{arg > 0 \land n = n} \quad IL001 : ldloc n
{arg > 0 \land s(0) = n} \quad IL002 : stloc name
{arg > 0 \land name = n \land 0 = 0} \quad IL003 : ldc 0
{arg > 0 \land name = n \land s(0) = 0} \quad IL004 : stloc balance

Figure 3.4: Bytecode proof example for initializing variables

{arg > 0} \quad IL001 : ldloc n
{arg > 0 \land s(0) = n} \quad IL002 : stloc name
{arg > 0 \land name = n} \quad IL003 : ldc 0
{arg > 0 \land name = n \land s(0) = 0} \quad IL004 : stloc balance

Figure 3.5: Simplified bytecode proof for 3.4
Algorithm 3.2 Simplification algorithm to eliminate equal subexpression such as \((x = x)\) in AND-Nodes

```plaintext
if AND_NODE.right.type = EQUAL then
    if AND_NODE.right.left = AND_NODE.right.right then
        AND_NODE ← AND_NODE.left
    end if
end if
if AND_NODE.left ≠ Null then
    AND_NODE.left.simplify
end if
if AND_NODE.right ≠ Null then
    AND_NODE.right.simplify
end if
```

3.2 Optimizing the Proof Script Generation

As explained in Section 2.1.2, the verification condition generator produces a list of proof obligations that need to be proven to show that the proof is correct. This Section presents two approaches how this proving of the proof obligations can be achieved. Subsection 3.2.1 explains how the proof checking was done initially by a simple script that uses the simplifier only. Subsection 3.2.2 talks about how the proof obligations can be proven much faster using an enhanced proof script. This optimization takes place at step B in Figure 3.1.

Example The following simple source program will be used to illustrate the differences between the two proving methods presented in the next Subsections.

```plaintext
1 identity (x: INTEGER): INTEGER
   require
   3 True
do
   5 { True }
   Result := x
   7 { True and Result = x}
ensure
   9 Result = x
end
```

Figure 3.6: Annotated Eiffel source program.

```plaintext
{ True } IL001 : ldloc x
{ True ∧ s(0) = x } IL002 : stloc Result
```

Figure 3.7: Simplified bytecode proof for source program in Figure 3.6.
3.2.1 Simple Proof Script Generation

A possibility for checking the proof obligations is to simply add every component of the Isabelle proof such as the variables, pre-/postconditions, instructions, method-bodies/-declarations, to the "simplifier". The simplifier then has to chose between a variety of possibilities which simplification to use when a certain statement needs to be simplified. This Section analyses this facility in more detail.

The simplifier is one of the central theorem proving tools in Isabelle [25]. In its most basic form, simplification means repeated application of equations from left to right. For example, taking the rules for $\oplus$ (see Appendix B.2 for details) and applying them to the term $\{0,1\} \oplus \{\}$ results in a sequence of simplification steps:

$$(0\oplus 1) \oplus \emptyset \quad \emptyset \oplus (1\oplus \emptyset) \quad \emptyset \oplus (\emptyset \oplus \emptyset) \quad \emptyset \oplus \emptyset$$

This is also known as term rewriting and the equations are referred to as rewrite rules. "Rewriting" would actually be more suitable than "simplification" because the terms do not necessarily become simpler in the process.

An Isabelle element is put on the simplifier by adding the attribute [simp] in front of its definition. The simplifier will then use it automatically. In addition, datatype and primrec declarations implicitly declare some simplification rules (see Appendix B.2 for an example). If the proof obligations should get proven by simplification solely, every constant definition of the Isabelle code would simply need the simplification attribute [simp] prepended.

The code below shows some selected parts of the generated Isabelle code for the example program. The normal and exceptional postcondition, the second instruction, the exception table and the method declaration elements were omitted. The focus of this Section lies on the usage of the [simp] attribute in the Isabelle theory. For a complete example of an Isabelle theory file see Chapter 6.

```isabelle
theory IDENTITY_PROOF_simp imports proof_checker_simp begin

... identity_x:: VarName [simp]:"identity_x \equiv 5"

identityPreCond:: Prec [simp]:"identityPreCond \equiv (\lambda s \sigma Z. True)"

...

P001identityBody:: Prec [simp]:"P001identityBody \equiv (\lambda s \sigma Z. True)"

L001identityBody:: InstSpec [simp]:"L001identityBody \equiv (P001identityBody, 001, (ldloc identity_x))"

...

identity_proof_body:: ClassBody [simp]:"identity_proof_body \equiv \{identity\}"
```
3.2.2 Enhanced Proof Script Generation

The proof script generation approach described in Section 3.2.1, which uses the simplifier only to solve the subgoals, is pretty simple. However, it has the disadvantage that the simplifier always tries to apply any rule in its simplification set for a rewriting. We therefore developed another approach of how the proof script can be generated. Instead of trying to apply every rule in its set, we give the simplifier a hint which rules to use. The measurements in Section 3.3 show that this enhanced proving method is much faster. This Subsection describes how the optimized proof script is generated.

Next to the simplification tool, Isabelle supports the notion of unfolding and folding definitions when proving a theorem. Unfolding has the same effect as applying the simplifier only for a specified definition, except that it acts on all subgoals, e.g. apply(simp only: xor_def). Folding is equivalent to the inverse of unfolding: the statements that match a specified definition get wrapped up back again to their initial definition. The following example illustrates how the folding and the unfolding methods work.

definition xor :: "bool ⇒ bool ⇒ bool" where
"xor A B ≡ ((A ∧ ¬ B) ∨ (¬ A ∧ B))"

Assuming we want to prove

lemma "xor A (¬ A)"

The lemma can be rewritten by unfolding some definitions:

apply(unfold xor_def)

In this case, the resulting goal

1. A ∧ ¬ A ∨ ¬ A ∧ ¬ A

can be proved by simplification. The lemma also could have been proved outright with apply(simp add: xor_def). This is okay for small and simple ex-
Proof Generation Optimizations

But as soon as the proof is composed out of several other components, the simplifier has too many rules in its simplification set and the proof checking will take much longer.

In the enhanced proof script approach described in this Section, we initially only put selected constant definitions of the Isabelle proof on the simplifier: definitions who merely serve as structure elements that make up the skeleton of the proof and do not directly have an influence on the proof outcome (e.g. the subtype set, variables etc.) Table 3.2.2 lists the constant definition elements that will be put on the simplifier initially and the elements that will be unfolded manually by the theorem proof script. The definitions which did not get on the simplifier will be rewritten in a specified order in the proof script using the unfold and fold methods. This makes sure, that the simplifier does not have to deal with too many choices when looking for a simplification.

<table>
<thead>
<tr>
<th>Isabelle Definition Element</th>
<th>On Simplifier</th>
<th>Unfolded by Script</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Instruction Precondition</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Exception Table</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Method Body</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Method Postconditions</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Method Precondition</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Method ID</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Method Declaration</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Variables</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Class Body</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Class Declaration</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>Subtype Set</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>CIL Program</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>

The Isabelle code for a CIL proof has to follow a certain structure to be readable by the proof checker. This Isabelle code structure is inherently predefined by the proof checker. Figure 4.4 in Section 4.2 illustrates which Isabelle constant definition elements are needed to constitute a valid proof and how the structure of such a proof should look like.

To show how the generated enhanced proof script folds and unfolds the selected Isabelle element definitions, we present the speed-optimized theorem proving script for the example program and analyze its generation step by step.

**Example**  The theorem `myProgram_theorem` states that the `is_program_safe` method is used to check the proof obligations for the CIL program `myProgram`, generated by the Verification Condition Generator `VCGen`. The definition of the `is_program_safe` statement and the Verification Condition Generator are listed in the Appendix B.1.

```isar
theorem myProgram_theorem: "(is_program_safe (VCGen myProgram))"
```

Since only the program, the subtype set, the class declarations and bodies, the method declarations and the variables are included in the simplifier, the program structure elements until the method bodies are rewritten using:

```isar
apply (simp)
```
After that, the method bodies and the Verification Condition Generator (VCGen) procedure for all methods need to be unfolded and the second, carried on CIL program should be folded back again by folding all method and class bodies and declarations and finally the program itself. This is needed, since the VCGen procedure for instructions VCGenInst that appeared when unfolding VCGenMethod still needs to get matched. For the example program, the following rule applications are generated:

apply (unfold identityBody_def)
apply (unfold VCGenMethod_base VCGenMethod_rec)
apply (fold identityBody_def)
apply (fold identity_def identity_proof_body_def)
apply (fold identity_proof_def myProgram_def)

Now, the method instructions for each method of all classes are unfolded. Furthermore, the proof checker definition VCGenInst_def and the contained assn_imp are rewritten:

apply (unfold L001identityBody_def L002identityBody_def)
apply (unfold VCGenInst_def assn_imp_def)

The proof structure is rewritten, such that the VCGenInstCaseAnalysis_def will match. The del in simp del makes sure that the listed definitions (VCGenInstCaseAnalysis_def, is_program_safe_base and is_program_safe_rec) will not be simplified. Again, the structure elements of the carried on program need to be folded back. For the example, the following script commands are generated.

apply (simp del: VCGenInstCaseAnalysis_def is_program_safe_base
is_program_safe_rec)
apply (fold identity_def identity_proof_body_def)
apply (fold identity_proof_def myProgram_def)

The VCGenInstCaseAnalysis_def distinguishes between the callvirtual, the throw and all the other instructions. In bigger programs, the unfolding of this case analysis can take some time. Again, the simp del ensures that we do not simplify too much. Furthermore, if a program contains any callvirtual instructions, the following generated script statements would also contain code for unfolding the id of the method being called.

apply (unfold VCGenInstCaseAnalysis_def)
apply (simp del: Wp_def is_program_safe_base is_program_safe_rec)

After that, all method bodies, their instructions and helper methods such as SuccL, getLabel and getPre are unfolded using the rule applications:

apply (unfold identityBody_def)
apply (unfold SuccL_base SuccL_rec getLabel_def getPre_def L001identityBody_def
L002identityBody_def)

Now, the case split is rewritten by the simplifier. The commands below also make sure that the Wp and the is_program_safe are not rewritten yet.
apply (simp del: Wp_def is_program_safe_base is_program_safe_rec)

The following step unfolds the weakest precondition production \( \text{Wp} \). This usually takes time, since the simplifier has to check several instructions that could match. Additionally, helper methods are unfolded and the whole subgoal gets simplified but without rewriting the \( \text{is\_program\_safe} \) definition yet. The following isabelle script commands are used to achieve this:

apply (unfold Wp_def)
apply (unfold SuccLabel_base SuccLabel_rec getLabel_def getPre_def)
apply (simp del: is_program_safe_base is_program_safe_rec)

Then, the instruction preconditions and method postconditions are rewritten using:

apply (unfold P001identityBody_def P002identityBody_def identityPostCond_def)

At this stage, only pure logical formulas are remaining. Assuming that the standard built-in theories of Isabelle [25] are imported, this should be no problem for the simplifier. If the proof contains a weak or a strong statement, the proof for the external weak or the strong theorem will be added to the simplifier in the last simplification step.

apply (simp del: is_program_safe_base is_program_safe_rec)
apply (simp)
done

Algorithm 3.3 describes informally how the optimized proof script for an Isabelle theory file, as presented in the example above, can be generated.
Algorithm 3.3 Informal description of the Algorithm for the optimized proof script generation

(* build up the proof structure *)
apply (the Simplifier)
apply (unfold all Method Bodies of all Class Bodies)
apply (unfold the VCGen Method definition)
apply (fold all Method Bodies and Declarations of all Class Bodies)
apply (fold all Class Bodies, Declarations and the Program)

(* unfold the instructions of the method bodies *)
apply (unfold all Instructions of all Methods and all Classes)
apply (unfold the VCGen Instruction definition and the Assertion Implication definition)

(* use the simplifier to remove the case split *)
apply (the Simplifier but without using the VCGen Instruction case analysis and the is_program_safe definitions)
apply (fold all Method Declarations of all Class Bodies and fold all Class Bodies)
apply (fold all Class Declarations and the Program)

(* unfold the instruction case analysis and simplify the case split but not the Wp *)
apply (unfold the VCGen Instruction case analysis definition)
apply (the Simplifier but without using the Weakest Precondition and the is_program_safe definitions)

if (the proof has callvirt instruction) then
    apply (unfold the Method ID of the callvirt Instruction)
    apply (the Simplifier but without using the Weakest Precondition and the is_program_safe definitions)
endif

(* unfold the method bodies and the helper methods *)
for all Method Bodies of all Class Declarations do
    apply (unfold the Method Body)
    apply (unfold the succLabel, the getLabel and the getPrec definitions and all Instructions of the current Method Body)
endfor

(* simplify the case split but not the Wp *)
apply (the Simplifier but without using the Weakest Precondition and the is_program_safe definitions)

(* now unfold the Wp and the helper functions *)
apply (unfold the Wp definition)
apply (unfold the succLabel, the getLabel and the getPrec definitions)
apply (the Simplifier but without using the is_program_safe definition)

(* unfold all instruction precondition and all method postcondition *)
apply (unfold all Instruction Preconditions and all Postconditions for all Methods of all Class Bodies)
Algorithm 3.4 Informal description of the Algorithm for the optimized proof script generation

(* at this step we should have only pure logical formulas remaining *)
apply (the Simplifier but without using the is_program_safe definition)

if (The proof has an additional implication proof) then
  apply (the Simplifier with the additional Lemmata)
else
  apply (the Simplifier)
endif

3.3 Measurements and Comparison

To approve the achievements of the optimized proof script, several measurements of the proof checking time have been made. In this Section, we present the results of the comparison of the two different proof scripts.

The measurements were made executing the Isabelle/HOL process in the bare-bone ML top-level mode, invoked from a text terminal under Cygwin [3]. The proof checker was loaded using the use_thy : string -> unit. The proof checking time was measured using the the flag time_use_thy : string -> unit when processing the theory file. The tests were carried out on a Intel Core 2 Duo CPU L7500 1.6Ghz computer with 2GB RAM.

Table 3.3 lists the Isabelle Theory Files used to measure the proof checking time. The column #Classes designates the number of classes in the theory file. The column #Methods denotes the number of methods in all processed classes of the checked Isabelle theory file. The values of column #Instructions are the summed up instructions of all methods of all classes of the checked Isabelle theory file. In the following, we shortly describe what the code of the different theory files does.

<table>
<thead>
<tr>
<th>Isabelle Theory File</th>
<th>#Classes</th>
<th>#Methods</th>
<th># Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDENTITY</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>OR ELSE</td>
<td>1</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>WRITE FIELD</td>
<td>1</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>LOOP</td>
<td>1</td>
<td>1</td>
<td>17</td>
</tr>
<tr>
<td>BANK ACCOUNT SIMPLER</td>
<td>1</td>
<td>3</td>
<td>18</td>
</tr>
<tr>
<td>BANK ACCOUNT</td>
<td>1</td>
<td>5</td>
<td>30</td>
</tr>
<tr>
<td>SUM INTEGERS</td>
<td>1</td>
<td>1</td>
<td>25</td>
</tr>
<tr>
<td>SUBTYPING</td>
<td>3</td>
<td>5</td>
<td>19</td>
</tr>
<tr>
<td>DEMO</td>
<td>4</td>
<td>8</td>
<td>84</td>
</tr>
</tbody>
</table>

Figure 3.8: Overview of the Isabelle theory files used to measure the proof checking time.
<table>
<thead>
<tr>
<th>Isabelle Theory File</th>
<th>Proof Checking Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Optimized Proof Script</td>
</tr>
<tr>
<td>IDENTITY</td>
<td>0.203</td>
</tr>
<tr>
<td>OR ELSE</td>
<td>0.922</td>
</tr>
<tr>
<td>WRITE FIELD</td>
<td>1.094</td>
</tr>
<tr>
<td>IF THEN ELSE</td>
<td>1.792</td>
</tr>
<tr>
<td>LOOP</td>
<td>3.157</td>
</tr>
<tr>
<td>BANK ACCOUNT SIMPLER</td>
<td>2.422</td>
</tr>
<tr>
<td>BANK ACCOUNT</td>
<td>4.531</td>
</tr>
<tr>
<td>SUM INTEGERS</td>
<td>6.282</td>
</tr>
<tr>
<td>SUBTYPING</td>
<td>2.313</td>
</tr>
<tr>
<td>DEMO</td>
<td>27.484</td>
</tr>
</tbody>
</table>

Figure 3.9: Comparison of the proof checking time for the optimized and the simple proof script.

**IDENTITY**  The method simply returns the given INTEGER argument.

**OR ELSE**  Assigns two BOOLEAN variables and tests their values with or else in the postcondition.

**WRITE FIELD**  Sets its own integer field to 2 and checks that the value was set.

**IF THEN ELSE**  Sets a local variable to the value of the method argument.

**LOOP**  Increments \( i \), initialized with 0, until \( i = 42 \).

**BANK ACCOUNT SIMPLER**  Bank account class that uses local variables for balance and name. Has methods for make, deposit and withdraw.

**BANK ACCOUNT**  Bank account class that uses a field for balance. Has methods make, deposit and withdraw as well as a setter and a getter method for the balance field.

**SUM INTEGERS**  Increments a local integer \( i \) until it reaches the value of the given argument. Uses a weak and a strong proof whose proof is in an external theory file that needs to get checked too. A full description of code can be found in Section 6.

**SUBTYPING**  Consists of a TEACHER and a STUDENT class inheriting from class PERSON. Reasons about the subtyping relationships using the is_equal and the conforms_to type functions.

**DEMO**  Consists of the classes BANK ACCOUNT, LOOP, IF THEN ELSE and SUM INTEGERS all put together in one program.

The values presented in Table 3.3 correspond to the arithmetically averaged times out of three proof checking runs for the respective Isabelle theory file. As the measurements show, using the optimized proof script speeds-up the proof checking for theory files with more than 18 instructions about a factor of 2.
Chapter 4

Design and Implementation

This chapter is about the design and implementation of the Isabelle code generator. Section 4.1 gives a general overview of the interface and the architecture of the code generator. Section 4.2 presents the theory file model used to generate code. Using the theory file model gives more flexibility during code generation and helps making sure that the generated theory file will be readable by the proof checker. Section 4.3 talks about the implementation of the visitors used to read the bytecode instructions and the bytecode proof expressions. Finally, Section 4.4 shows how the Isabelle code for inheritance sets, types and weak and strong proofs is generated. The Isabelle code generator is written in Eiffel [20, 4] and has been integrated in the Proof-Transforming Compiler environment of the Paco [7] project.

\[\text{Figure 4.1: Overview of the information flow in connection with the PTC}\]

4.1 Overview

The Isabelle code generator presented in this thesis is able to interpret and process the bytecode proof output of the Proof-Transforming Compiler developed by Nordio, Karahan, Guex, Hess and Hauser [30, 19, 17, 18]. The Isabelle code
generator takes an Abstract Syntax Tree (AST) of a bytecode proof as input, creates the corresponding Isabelle code and stores the resulting output in a STRING object. The processing can be continued by passing on the output to a proof checker. The code generator is used for the embedding of a bytecode proof to a proof checker in the context of a Proof-Carrying Component framework. Figure 4.1 illustrates an overview of this setting.

4.1.1 Code Generator Interface

The interface of the code generator is pretty small. A client of the Isabelle code generator only needs to know how to initialize the generator, how to invoke the code generation, where to fetch the result and how to change the generation settings. Figure 4.2 shows the exported features of the Isabelle code generator.

![Figure 4.2: Overview of the Isabelle code generator interface](image)

The code generator is initialized by giving a bytecode proof AST of type BC_PROOF produced by the Proof-Transforming Compiler as argument. The code generation is then invoked by the feature generate_isabelle_file. After that, the produced Isabelle code output gets stored in the isabelle_output object reference of type STRING. Additionally, further methods allow to change the behavior of the code generation. The feature set_generate_enhanced_proofs for example allows to chose between the generation of a simple or an enhanced proof script. The simple and enhanced proof scripts are described in Section 3.2.

4.1.2 Code Generation Workflow

The code generation is initiated by calling the exported feature generate_isabelle_file. Invoking the feature triggers the following steps:

1. Simplification of the bytecode contract expressions
2. Visitation of the bytecode proof
3. Generation of possible extra files (weak/strong proofs)

The first step only changes the bytecode proof contract expressions and does not generate any output yet. This bytecode proof simplification process is described in Section 3.1. During the second step though, the main Isabelle theory file is constructed, while visiting the bytecode proof, and written to the result object. Section 4.3 presents the visitors used to read the bytecode proof. The third step is only executed when a weak or a strong proof construct was detected during the second step. If the bytecode proof contained such a construct, an additional Isabelle theory file is written to the output directory. Section 4.4.1 describes this step in more detail.
When visiting the bytecode proof elements, the code generator does not directly append the corresponding Isabelle code to the output object. Instead, the code generator first builds up an internal theory file model of the Isabelle proof. The Isabelle code for the respective element is then generated just after enough information for the theory file model element (e.g. for the classbody element) has been gathered. The generated code is stored in the theory file model element and can be recalled from there. The Isabelle code for the whole proof can be retrieved by calling `out` on the root element (the Isabelle `theory` element). As an example, Figure 4.3 shows the `out` feature of the `theory` element.

```isabelle
feature -- Result

out: STRING is
  -- string representation of the generated isabelle theory
  do
    Result := header.out + program.out
  end
```

Figure 4.3: Feature `out` of the class `ISABELLE_THEORY`

The `out` feature of each theory file model element defines how its own element code and the code of its subelements are mixed together to make up the output.

### 4.1.3 Naming Conventions

Every Isabelle constant definition in an Isabelle theory file needs to have a unique name such that it can be processed by the proof checker. In this thesis, we assume that the given bytecode proof transformed by the Proof-Transforming Compiler does not have any classes, methods or variables with the same name. The names for constant definitions for theory file model elements can therefore be composed by simply using the corresponding bytecode element name and combining it with further lexical scope information, e.g. by pre- or postpending it with the method and/or class name if necessary. The following constant definition is an example of an instruction specification element for the bytecode instruction `IL002` of the method `identity`. The name of the Isabelle instruction specification equivalent `L002identityBody:: InstSpec` is built by appending the method body name `identityBody` to the instruction number `L002`.

```isabelle
constdefs
L002identityBody:: InstSpec
  "L002identityBody ≡ (P002identityBody, 002, (stloc RESULT))"
```

### 4.2 Isabelle Theory File Model

When visiting the bytecode proof elements, the code generator also creates corresponding theory file model elements. These model elements serve as a temporary storage for generated code and as an information base for the code generation of further elements. The usage of this theory file model has several advantages.
The additional level of indirection introduced through the utilization of the theory file model results in much a higher flexibility during the code generation process: Changes concerning the order of the code components (constdefs elements) in the theory file can be made at any time. The feature `out` defines which subelements and in which order the subelements should be added to the output object. When detecting a method call instruction, this added feature comes in handy. Since it is necessary, when generating code for method calls, to know the typename and the method ID in advance, the easiest approach to solve this problem is to prepend the Isabelle theory file with all the typename and method ID declarations. This enables to talk about the components later in the file without having to give their concrete definition yet.

Furthermore, the introduction of the theory file model can also be used for lexical scoping information. This is very useful, if for example, a unique ID for a certain element has to be created. Most of the theory model elements have a link to their parent element from which they can fetch the information needed or go on to other elements. This is similar to compiler design where a symbol table datastructure serves as a base for this kind of information.

![BON Diagram for the Isabelle Theory File Model Classes](image)

Figure 4.4: BON Diagram for the Isabelle Theory File Model Classes

The theory file also helps when generating constructs such as the subtype-set or when adjusting the `imports` directive to include additional files (weak and strong proofs) in the preprocessing phase. Furthermore, the stored information in the theory model is also used to generate the optimized proof script. Using the theory model avoids to be forced to do a new visit of the bytecode proof. Briefly worded, the following software developer wisdom concisely describes the reason for introducing the theory model:

*There is no software engineering problem that cannot be solved by adding another layer of indirection* [1].

Figure 4.4 illustrates the design of the Isabelle theory file model used in this thesis. The deferred class `ISABELLE_ELEMENT` summarizes the base
functionality of every theory model element. Every model element has a name, an output feature and an internal buffer to store the respective Isabelle code. Figure 4.5 gives a structural overview of the Isabelle theory model. A connection between two blocks in the Figure corresponds to a has-a relationship between two model elements. The code element for the subtype set is not in the Figure 4.5 since it is generated by a special code generator, described in Section 4.4.1.

![Figure 4.5: Structure of the Isabelle Theory File Model Elements](image)

### 4.3 Tree Visitors

The information from a bytecode proof is read using the visitor pattern [15]. The contract simplifier 4.3.1 and the skeleton visitor 4.3.2 process the bytecode skeleton elements `BC_PROOF`, `BC_CLASS_PROOF`, `BC_METHOD_PROOF` and `BC_FIELD`. Both visitors are descendants of the deferred class `SKELETON_VISITOR_DEF`.

Subsection 4.3.1 describes the implementation details of the contract simplifier and Subsection 4.3.2 presents the implementation of the skeleton visitor.
The contract expressions of a bytecode instruction are processed using a contract visitor, described in Subsection 4.3.3, and the instructions are handled using an instruction visitor as in Subsection 4.3.4.

4.3.1 Bytecode Proof Simplification Visitor

A bytecode proof simplification visitor of class `CONTRACT_SIMPLIFIER` implements the algorithmic properties of the bytecode proof simplification optimizations described in Section 3.1. The visitor loops over all instructions of a `BC_METHOD_PROOF` class and invokes the feature `simplify` on the instruction preconditions. A simplification algorithm can be added by implementing the `simplify` feature in a descendant of class `AST_BINBOOLEXP`. The `EQUAL-Node` simplification presented in Section 3.1.1 is implemented in the `simplify` feature of class `VIS_EQ` and the `AND-Node` simplification of Section 3.1.2 in the `simplify` feature of class `VIS_AND`.

4.3.2 Skeleton Visitor

The skeleton visitor implements the formalization of the embedding of classes, methods and attributes as described in Section 2.2.1. The skeleton visitor generates the following theory model elements while visiting the bytecode proof components:

- The `visit_proof` feature creates the general isabelle theory header, the program header, generates the subtype set and changes the import directive if an additional Isabelle file has to be preprocessed.

- The `visit_class_proof` feature creates the class declaration, -body and -type elements.

- The `visit_method_proof` feature creates the method declaration, -body and -ID elements, the exception table, variables for locals and arguments and creates the method pre- and postcondition elements with default values (`True` for the precondition and the normal postcondition and `False` for the exceptional postcondition). The default values will be overwritten by the `CONTRACT_VISITOR` if their values were defined in the bytecode proof.

- The `visit_field` creates the field statement, the field declaration and -type.

4.3.3 Contract Visitor

The contract visitor implements the expression translations formalized by the translation function $\nu_E$ in Section 2.2.2. The visitor `CONTRACT_VISITOR` implements the deferred class `DEF_VISITOR` and uses the type generator described in Section 4.4.3 to generate the correct Isabelle types for the contract expressions. The visitor gets used to embed bytecode method pre- and postconditions, instruction preconditions and expressions for weak and strong proof implications to Isabelle. Table 4.1 gives an overview of the expressions the contract visitor in class `CONTRACT_VISITOR` is able to embed to Isabelle.
**4.3.4 Instruction Visitor**

The instruction visitor implements the instruction translations formalized by the translation function $\nabla_S$ in Section 2.2.3. The instruction visitor is a descendant of class `INSTRUCTION_VISITOR_DEF`. The instructions that are translated using a deep embedding are listed in Table 4.2.

The `visit_try_catch` feature does not translate to Isabelle code directly. Instead, an exception table entry for the detected try-catch statement is added to the theory model exception element. An eventual `throw` instruction within the `try_catch` is then caught and the control is handed to the corresponding exception table entry.
The \textit{visit\_nop} is also used to encode information about an attached strong or weak proof. If the query \textit{nop\_has\_proof} yields \textit{True}, an additional Isabelle theory file is constructed using the weak and strong proof generator described in Section 4.4.1.

<table>
<thead>
<tr>
<th>CIL Instruction</th>
<th>Visitor feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{br}</td>
<td>\texttt{visit_br}</td>
</tr>
<tr>
<td>\texttt{brtrue}</td>
<td>\texttt{visit_brtrue}</td>
</tr>
<tr>
<td>\texttt{brfalse}</td>
<td>\texttt{visit_brfalse}</td>
</tr>
<tr>
<td>\texttt{callvirt}</td>
<td>\texttt{visit_callvirt}</td>
</tr>
<tr>
<td>\texttt{ldc}</td>
<td>\texttt{visit_ldc}</td>
</tr>
<tr>
<td>\texttt{ldloc}</td>
<td>\texttt{visit_ldloc}</td>
</tr>
<tr>
<td>\texttt{ldfld}</td>
<td>\texttt{visit_ldfld}</td>
</tr>
<tr>
<td>\texttt{ret}</td>
<td>\texttt{visit_ret}</td>
</tr>
<tr>
<td>\texttt{stloc}</td>
<td>\texttt{visit_stloc}</td>
</tr>
<tr>
<td>\texttt{stfld}</td>
<td>\texttt{visit_stfld}</td>
</tr>
<tr>
<td>\texttt{newobj}</td>
<td>\texttt{visit_newobj}</td>
</tr>
<tr>
<td>\texttt{iladd}</td>
<td>\texttt{visit_and}</td>
</tr>
<tr>
<td>\texttt{ilor}</td>
<td>\texttt{visit_or}</td>
</tr>
<tr>
<td>\texttt{ilceq}</td>
<td>\texttt{visit_ceq}</td>
</tr>
<tr>
<td>\texttt{ilcgt}</td>
<td>\texttt{visit_cgt}</td>
</tr>
<tr>
<td>\texttt{ilcge}</td>
<td>\texttt{visit_cge}</td>
</tr>
<tr>
<td>\texttt{ilctime}</td>
<td>\texttt{visit_clt}</td>
</tr>
<tr>
<td>\texttt{ilclt}</td>
<td>\texttt{visit_clt}</td>
</tr>
<tr>
<td>\texttt{ilcneq}</td>
<td>\texttt{visit_cneq}</td>
</tr>
<tr>
<td>\texttt{iladd}</td>
<td>\texttt{visit_add}</td>
</tr>
<tr>
<td>\texttt{ilsup}</td>
<td>\texttt{visit_sub}</td>
</tr>
<tr>
<td>\texttt{ildiv}</td>
<td>\texttt{visit_div}</td>
</tr>
<tr>
<td>\texttt{ilmul}</td>
<td>\texttt{visit_mul}</td>
</tr>
<tr>
<td>\texttt{nop}</td>
<td>\texttt{visit_nop}</td>
</tr>
<tr>
<td>\texttt{throw}</td>
<td>\texttt{visit_throw}</td>
</tr>
</tbody>
</table>

Table 4.2: Instructions embedded by the class \textit{INSTRUCTION\_VISITOR}

### 4.4 Special Generators

Next to the three Isabelle code generating visitors described in Section 4.3, a special code generator for handling the weak and strong proofs of a bytecode proof has been built. Subsection 4.4.1 describes how the support for weak and strong proof Isabelle code was implemented. Aside from that, the inheritance set generation was put in a separate class to be able to use it in a more modular fashion compared to generating the code directly in one of the vistors. Section 4.4.2 presents the details about the inheritance set generation class. Subsection 4.4.3 concludes the special generators section by presenting how the types for Isabelle expressions are created.
4 Design and Implementation

4.4.1 Weak and Strong Proofs

This Section describes how the Isabelle code for the language-independent weak and strong rules is generated. As described by Nordio, Müller and Meyer in [29], in a strength transformation, \( P' \implies P \) and \( P \ s_1 \ Q \) are translated using the nop instruction, where \( P \) denotes the precondition, \( Q \) the postcondition (predicates in first order logic) and \( s_1 \) the statement. A weak rule translates \( Q \implies Q' \) respectively. For details concerning the translations see [29].

Figure 4.6 shows bytecode proof code snippets for a weak and a strong rule. The complete annotated bytecode proof and the corresponding generated Isabelle code can be found in Chapter 6.

\[
\begin{align*}
\{ \ n > 1 \text{ and Result } = 1 \text{ and } i = 2 \} \\
\Rightarrow \{ \text{Weak Rule, Isabelle Proof 1} \} \\
\{ \text{Result } = ((i-1)*i)/2 \text{ and } n+1 \geq i \text{ and } i > 1 \} \\
... \\
\{ \ n \neq n+1 \text{ and Result } = ((i-1)*i)/2 \text{ and } n+1 \geq i \text{ and } i > 1 \} \\
\Rightarrow \{ \text{Strength Rule, Isabelle Proof 2} \} \\
\{ \ n \neq n+1 \text{ and Result } + i = ((i(i+1))/2) \text{ and } n+1 \geq i \text{ and } i > 1 \} 
\end{align*}
\]

Figure 4.6: Bytecode proof snippets for a weak and a strong rule

The difficulty lies in proving that \( P' \implies P \) and \( Q \implies Q' \). The information how to prove these weak and the strong implications is delivered in the additional files IsabelleProof1 and IsabelleProof2 respectively. Figure 4.8 and Figure 4.7 present the additional files for the example bytecode proof snippets in Figure 4.6. An additional proof file can either contain a simple PROOF: element as presented in Figure 4.8 or consist out of a proof and auxiliary lemmata that can be used in the PROOF: element. The additional proof code in Figure 4.7 shows an example of such a file.

\[\text{AUXILIARY LEMMATA NAMES:}\]
\text{sum\_i\_eq}

\[\text{AUXILIARY LEMMATA PROOFS:}\]
\text{lemma sum\_i\_eq: } "((i-(1::int))*i) div 2)+i = (i*(i+1) div 2)"
\text{apply(simp add: ring\_simps)}
\text{apply(arith)}
done

\[\text{PROOF:}\]
\text{apply(simp add: sum\_i\_eq)}

Figure 4.7: Isabelle Proof 2 for strong rule in Figure 4.6
The class \texttt{AUXILIARY\_LEMMA\_READER} implements a reader for such additional Isabelle implication proof files. The implication proofs read by the reader are passed on to the \texttt{ISABELLE\_CPX\_IMPLICATION\_GENERATOR} class which builds an Isabelle theory file for the implication proofs. The complex implication generator class uses a contract visitor as described in 4.3.3 to embedd the bytecode implication expressions to a valid Isabelle implication expression.

### 4.4.2 Inheritance Set

To be able to reason about subtyping in Isabelle, a set specifying all class relationships in a program is used. Membership of the set implies a subtyping relationship. The example subtyping set in Figure 4.9 depicts the subtyping relationships: \texttt{TEACHER \preceq PERSON} and \texttt{STUDENT \preceq PERSON}. The subtype set resulted form the transformation of a sample Eiffel program. Since Eiffel supports multiple inheritance, the Proof-Transforming Compilation of such classes is non-trivial [29]. An Eiffel type is always mapped to a CIL interface class, e.g. \texttt{PERSON\_INTERFACE\_type} and a CIL implementation class, e.g. \texttt{PERSON\_type}. This blows up our subtyping set a little.

\begin{verbatim}
defs cil_subtypeSet_def[simp]:"cil\_subtypeSet \equiv
type (TEACHER\_type,TEACHER\_INTERFACE\_type),
(type (PERSON\_INTERFACE\_type,PERSON\_INTERFACE\_type),
(type (PERSON\_type,PERSON\_INTERFACE\_type),
(type (STUDENT\_type,STUDENT\_INTERFACE\_type),
(type (STUDENT\_INTERFACE\_type,PERSON\_INTERFACE\_type)"
\end{verbatim}

Figure 4.9: Isabelle code for an example subtype set

The generation of the inheritance set such as in Figure 4.9 is controlled by the class \texttt{ISABELLE\_SUBTYPE\_SET\_GENERATOR}. When visiting the bytecode classes \texttt{BC\_CLASS\_PROOF} of a program, the visitor also looks for subtyping relationships. If an inheritance relation is detected, a new tuple with the two relevant classes is added to the subtype set using the feature \texttt{add\_subtype\_relation}. The generated subtype set Isabelle code is then fetched from the \texttt{out} feature of the Isabelle program element by calling the query \texttt{subtype\_set} of the \texttt{ISABELLE\_SUBTYPE\_SET\_GENERATOR} object reference.

### 4.4.3 Isabelle Types

Isabelle uses a typed logic whose type system resembles that of functional programming languages like ML or Haskell [25]. It supports the following built-in base types:
• Base types such as \texttt{bool}, the type of truth values and \texttt{nat}, the type of natural numbers

• Type constructors such as \texttt{list} the type of lists and \texttt{set} the type of sets

• Function types denoted by $\Rightarrow$

• Type variables denoted by 'a, 'b etc.

Using types prevents users from writing nonsense program code. Isabelle insists that all terms and formulas are well-typed and prints an error message if a type mismatch is encountered. To reduce the amount of explicit type information that needs to be provided by the user, Isabelle tries to infer the type of all variables automatically (type inference). Because this may lead to misunderstandings, Isabelle can be told to print out type information explicitly using the command: ML \\\texttt{"set show types"}.  

\textbf{Algorithm 4.1} Type information generation algorithm for binary-operation expressions such as AND ($\land$), OR ($\lor$) etc.

\begin{verbatim}
left_type ← ""
right_type ← ""
if node.left.type ≤ VIS_ID ∨ node.left.type ≤ VIS_STACK then
  left_type ← "aBool"
else if node.left.type ≤ VIS_EXP then
  if node.left.exp.type ≤ VIS_ID then
    left_type ← "aBool"
  end if
end if
if node.right.type ≤ VIS_ID ∨ node.right.type ≤ VIS_STACK then
  right_type ← "aBool"
else if node.right.type ≤ VIS_EXP then
  if node.right.exp.type ≤ VIS_ID then
    right_type ← "aBool"
  end if
end if
\end{verbatim}

However, when working with values on the (untyped) stack, Isabelle is not able to infer the correct type. We therefore decided to provide the type information for the generated Isabelle code explicitly. This allowed us to solve the type information problem more systematically. But nevertheless, this approach might be improved (see Chapter 7.2). Algorithm 4.1 describes which types for binary-operation expressions are generated and Algorithm 4.2 shows how the types for equal (=) and not-equal (\neq) expressions are generated. If not set by the algorithm, the resulting type \texttt{left_type} or \texttt{right_type} will stay empty, meaning that no type information is constructed.
Algorithm 4.2 Type information generation algorithm for EQUAL (=) and NOTEQUAL (\(\neq\)) expressions

\[
\begin{align*}
\text{left\_type} & \leftarrow "" \\
\text{right\_type} & \leftarrow ""
\end{align*}
\]
{Stack = Contract/Expression}

if \(\text{node.left.type} \leq \text{VIS\_STACK}\) then

{Stack = Contract}

if \(\text{node.right.type} \leq \text{VIS\_CONTRACT}\) then

if \(\text{node.right.type} \leq \text{AST\_BIN\_BOOLEXP} \lor \text{node.right.type} \leq \text{AST\_UNARY\_BOOL\_VALUE}\) then

\(\text{right\_type} \leftarrow \text{"boolV"}\)

else if \(\text{node.right.type} \leq \text{VIS\_EXP}\) then

if \(\text{node.right.type} \leq \text{VIS\_TERM} \lor \text{node.right.type} \leq \text{VIS\_FACT}\) then

\(\text{left\_type} \leftarrow \text{"aInt"}\)

\(\text{right\_type} \leftarrow \text{"aInt"}\)

end if

else

\(\text{left\_type} \leftarrow \text{"aInt"}\)

\(\text{right\_type} \leftarrow \text{"aInt"}\)

end if

else if \(\text{node.right.type} \leq \text{AST\_BIN\_BOOLEXP} \lor \text{node.right.type} \leq \text{AST\_UNARY\_BOOL\_VALUE}\) then

\(\text{left\_type} \leftarrow \text{"aBool"}\)

else if \(\text{node.right.type} \leq \text{VIS\_INTEGER} \lor \text{node.right.type} \leq \text{VIS\_STACK} \lor \text{node.right.type} \leq \text{VIS\_TERM} \lor \text{node.right.type} \leq \text{VIS\_FACT} \lor \text{node.right.type} \leq \text{VIS\_EXP}\) then

\(\text{left\_type} \leftarrow \text{"aInt"}\)

\(\text{right\_type} \leftarrow \text{"aInt"}\)

end if

end if

else if \(\text{node.left.type} \leq \text{VIS\_ID} \lor \text{node.left.type} \leq \text{VIS\_EXP}\) then

if \(\text{node.right.type} \leq \text{AST\_BIN\_BOOLEXP} \lor \text{node.right.type} \leq \text{AST\_UNARY\_BOOL\_VALUE}\) then

\(\text{right\_type} \leftarrow \text{"aBool"}\)

else if \(\text{node.right.type} \leq \text{VIS\_INTEGER}\) then

\(\text{left\_type} \leftarrow \text{"aInt"}\)

\(\text{right\_type} \leftarrow \text{"aInt"}\)

else if \(\text{node.right.type} \leq \text{VIS\_EXP}\) then

if \(\text{node.right.exp.type} \leq \text{VIS\_ID} \lor \text{node.right.exp.type} \leq \text{VIS\_INTEGER} \lor \text{node.right.exp.type} \leq \text{VIS\_TERM} \lor \text{node.right.exp.type} \leq \text{VIS\_FACT}\) then

\(\text{left\_type} \leftarrow \text{"aInt"}\)

\(\text{right\_type} \leftarrow \text{"aInt"}\)

end if

end if
end if

end if
Chapter 5

Extending Proof-Transforming Compilation

In Chapter 2, we presented the type functions conforms to and is_equal for Eiffel contracts, which reason about subtyping relationships. However, to be able to write meaningful applications of those type functions, using one single class is not sufficient. Different classes with inheritance relationships need to be provided such that reasonable contracts can be expressed. Initially, the Proof-Transforming Compiler interface in EiffelStudio was only able to process one class.

This Chapter presents the extension applied to the Proof-Transforming Compiler interface in EiffelStudio, such that working with several classes is now possible. Furthermore, we describe how we added a new EiffelStudio interface for generating the Isabelle code for a bytecode proof produced by the Proof-Transforming Compiler.

The extensions build upon the existing EiffelStudio interface for the Proof-Transforming Compiler implemented by Hess [18]. Section 5.1 talks about how we added support for invoking the Proof Transforming Compiler for a cluster of classes and Section 5.2 illustrates the design of the added interface to the Isabelle code generator. Finally, the Chapter concludes with summarizing different further changes and extensions applied to the Proof-Transforming Compiler in Section 5.3.

5.1 Cluster Generation

To ensure continuity, we decided to stick to the existing Proof Transforming Compiler interfacing design by Hess [18]. The extension for invoking the Proof Transforming Compiler for clusters is implemented in the same style: To generate the bytecode proof for several annotated Eiffel source program classes, one simply needs to pick a cluster from the Cluster panel and drop the picked cluster to the CIL Proof Tool panel. Dropping the cluster invokes the Proof Transforming Compiler for all classes in the cluster and adds each transformed bytecode
class to the \texttt{BC\_PROOF} reference (instead of creating a \texttt{BC\_PROOF} for each class as before). Figure 5.1 shows an example output to the CIL Proof Tool panel after the Proof-Transforming Compiler has been invoked for a cluster of classes.

![Example output after invoking the Proof-Transforming Compiler for a cluster of classes.](image)

The support for cluster generation was implemented by adding a new \textit{agent drop cluster} to the list of drop actions for the CIL Proof Tool. To synchronize the information between the CIL Proof and the Eiffel Proof Tool, the Proof-Transforming Compiler interface was implemented using the Observer Pattern [15]. To avoid too much overhead when generating the transformed bytecode proof for a cluster of classes, we added a new feature for invoking the Proof-Transforming Compiler without notifying its Observers.

### 5.2 Interfacing the Isabelle Generator

The Isabelle code generator presented in this thesis has a small and clean interface. The generator takes a \texttt{BC\_PROOF} as input and presents the generated Isabelle code as output in a \texttt{STRING} object. The interface makes an interfacing to EiffelStudio a simple task. We added a new button \textit{Generate Isabelle Theory} to the CIL Proof Tool panel and defined the \textit{agent generate_isabelle_theory} to be invoked when a push action occurred.

The feature \textit{generate_isabelle_theory} brings up a question dialog which asks whether a simple or an optimized proof script should be generated along with the Isabelle code. The user can chose one of the two options and after the selection, the feature \textit{set_generate_enhanced_proof_commands} on the Isabelle code generator reference will be invoked with either \texttt{true} or \texttt{false} as the setter methods argument. Figure 5.2 shows the question dialog that pops up before the Isabelle code is generated with the corresponding proof script.
After having chosen the generation option, the Isabelle code is generated and written to the output file in the `isabelle_file_directory.name` directory. The exact location and the filename are written to the CIL Proof Tool panel, as well as information about the progress of the Isabelle code generation. Figure \ref{fig:isabelle_code_generation} shows an example output of a successful Isabelle theory file generation.

egin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{isabelle_code_generation.png}
\caption{Output after successful generation of the Isabelle code.}
\label{fig:isabelle_code_generation}
\end{figure}

Next to the Isabelle theory file generation we also added an interface for simplifying the bytecode proof produced by the Proof-Transforming Compiler. We added another button to the CIL Proof Tool panel with the name `Simplify Proof`. When the button is pushed, the registered feature `show_bytecode_proof_simplified` is invoked. This action feature then visits the current `bytecode_proof`, simplifies its contracts according to the algorithms described in Section \ref{sec:proof_transforming} and replaces the `BC_CLASS_PROOF` references with the simplified ones.

Figure \ref{fig:simplify_bytecode} shows the interface buttons for the bytecode proof simplifier and the Isabelle code generator.

egin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{simplify_bytecode.png}
\caption{EiffelStudio interface for simplifying the bytecode proof with button `Simplify Proof` and generating the Isabelle theory file with button `Generate Isabelle Theory`.}
\label{fig:simplify_bytecode}
\end{figure}
5.3 PTC Implementation Changes

External Proof for Weak and Strong Rule Implications. The weak and the strong rule, described by in detail in Nordio [29], need a proof for the weakening, respectively the strengthen implication. The reference to the proof must be introduced in the stage before processing the annotated source proof by the Proof-Transforming Compiler and it must be carried on to every succeeding transformation. If not, the additional proof information is not available to the proof checker and not all subgoals can be proved - the proof will be rejected.

In this thesis, we extended the weak and strong proof XML elements with an additional isabelle subelement in the implication element. The file specified in the isabelle XML element holds an external proof for a weak or a strong rule, respectively. Figure 5.5 shows an example fragment of an XML weak proof element.

```xml
<weak-proof ID="weak">
  <precondition></precondition>
  <postcondition>
    <normal></normal>
  </postcondition>
  <s1 ref="init-i">
    <implication mode="normal">
      <lhs></lhs>
      <rhs></rhs>
      <isabelle>my_weak_proof.txt</isabelle>
    </implication>
  </s1>
</weak-proof>
```

Figure 5.5: Example fragment of an XML weak proof element.

Furthermore, we extended the PIT_WEAK and the PIT_STRONG classes such that they also transform the link to the external proof. The created NOP instruction during the transformation holds an additional field for the weak or strong proof. Section 4.4.1 describes how the proofs from the external file are embedded to Isabelle.

Inheritance. As described in the introduction of this Chapter 5, the Proof-Transforming Compiler lacked support for working with several classes. Next to that, there was also no possibility to describe inheritance relationships between classes on the source code level. We extended the XML class definition and the Proof-Transforming Compiler to support inheritance declarations to overcome this problem. Figure 5.6 shows a fragment of an example XML class proof containing an inheritance relationship element. On the bytecode level, we extended the BC_CLASS_PROOF class with a field parent which declares the name of the parents the actual class inherits from. The inheritance information can then be used by the Isabelle code generator to build the Isabelle inheritance set as described in Section 4.4.2.

Check Proof. In order to test our embedding of the throw bytecode instruction and the Isabelle exception table, we implemented the check instruction
translation defined by Nordio [29]. The check translation was implemented in class PIT_CHECK_PROOF.

To translate a check instruction, first the expression $e$ that needs to be checked is pushed on top of the stack. If $e$ evaluates to true, the control is transferred to the next instruction. Otherwise, an exception is thrown by putting a new exception object on top of the stack and giving control to the throw instruction. For a description of the translation definitions and other details see [29]. The check translation is specified as follows:

\[
\nabla S \left( \{ P \} \text{check} e \text{end} \{ (P \land e) \land \text{false}, (P \land \neg e) \} \right) = \nabla E( P, e, shift(P) \land s(0) = e, l_a ) \\
\text{shift}(P) \land s(0) = e \quad l_a : \text{brtrue} \quad l_{\text{next}}
\]

\[
\begin{align*}
P \land \neg e \quad l_e : \text{newobj} \text{Exception()} \\
P \land \neg e \land s(0) = \text{new}($ & \text{Exception) } l_d : \text{throw}
\end{align*}
\]
Chapter 6

Applications

This chapter presents an example application of the Proof-Carrying Components infrastructure. An example Eiffel source program is fed as input to the Proof-Carrying Components framework. We show the code output after each transformation step. An overview of the setting is illustrated in Figure 1.1 of the Introduction Chapter 1.

The example program implements a function that returns the sum from 1 to \( n \). The source code of the program is presented in Figure 6.1 right below. Figure 6.2 shows the proof annotated Eiffel program and Figure 6.3 presents the bytecode proof produced by the Proof-Transforming Compiler for the input source proof program in Figure 6.2. Finally, Figure 6.4 and Figure 6.5 show the generated Isabelle code for the bytecode proof. The generated Isabelle code can then be checked by the proof checker.

```eiffel
1 sum (n: INTEGER): INTEGER
   require
   positive : n>1
   local
   i: INTEGER
   do
      from
      Result := 1
      i := 2
      invariant
      Result = ((i-1)*i)/2 and n+1 >= i and i > 1
      until
      i = n+1
      loop
      Result := Result + i
      i := i+1
   end
   ensure
   Result = (n*(n+1))/2
end
```

Figure 6.1: Eiffel source code for a function returning the sum from 1 to \( n \)
Annotated Source Proof. To prove the source program in Figure 6.2, we applied the assignment, loop, and compound rule as well as the weak and the strength rules of the logic for Eiffel as presented in [29]. The most interesting part of the proof is the application of the weak and the strength rule:

Lines 13-15 state that the invariant of the loop holds at the entry of the loop using the weak rule. To proof this implication, we wrote a proof in Isabelle (the proof is straightforward: apply auto).

Lines 21-23 show that the loop invariant implies the weakest precondition of the loop body by applying the strength rule. We wrote another proof in Isabelle to show that this implication is true. The Isabelle proof for the strength rule is more complex, since it needs an auxiliary lemma and the Isabelle arith tactic.

```plaintext
sum (n: INTEGER): INTEGER
2  require
3     positive : n>1
4  local
5     i: INTEGER
6  do
7      { n > 1 }
8  from
9    Result := 1
10    { n > 1 and Result = 1 }
11    i := 2
12    { n > 1 and Result = 1 and i = 2}
13    ⇒ [Weak Rule, Isabelle Proof 1]
14    { Result = ((i−1)*i)/2 and n+1 ≥ i and i > 1 }
15 invariant
16    Result = ((i−1)*i)/2 and n+1 ≥ i and i > 1
17 until
18    i = n+1
19 loop
20      { i ≠ n+1 and Result = ((i−1)*i)/2 and n+1 ≥ i and i > 1 }
21      ⇒ [Strength Rule, Isabelle Proof 2]
22      { i ≠ n+1 and Result + i = (n(i+1))/2 and n+1 ≥ i and i > 1 }
23      Result := Result+i
24      { i ≠ n+1 and Result = (i*(i+1))/2 and n+1 ≥ i and i > 1 }
25      i := i+1
26    { Result = ((i−1)*i)/2 and n+1 ≥ i and i > 1 }
27 end
28 ensure
29    Result = (n*(n+1))/2
30 end
```

Figure 6.2: Proof-Annotated program for the source example in Figure 6.1
Bytecode Proof. The result of the proof transformation of Figure 6.2 is shown in Figure 6.3. The translation of default type initialization produces lines $IL001$-$IL004$. Lines $IL005$-$IL008$ are produced by the translation of the body of the from part of the loop. The weak and strength rules are translated in lines $IL009$ – $IL011$. Lines $IL012$ – $IL019$ translate the loop body and the until expression is translated in lines $IL020$ – $IL025$.

To show that the bytecode proof is a valid proof, it is necessary to show that each precondition implies the weakest precondition of the successor instruction. In most of the cases, this proof can be done by applying the definition of weakest precondition. The translation of the weak and the strength rule in lines $IL009$ – $IL010$ and lines $IL011$ – $IL012$ respectively are the most interesting cases in the bytecode proof shown in Figure 6.3. The two implications are proven since the proofs have been developed for the source program, and these proofs are translated to CIL. Thus, the translation introduces two lemmata that show the precondition at $IL009$ implies the precondition at $IL010$, and the precondition at $IL011$ implies the precondition at $IL012$.

Generated Isabelle Code. The generated Isabelle code for the bytecode proof in Figure 6.3 is illustrated in Figure 6.4. The Isabelle code has been generated by using the optimized proof script generation as described in Chapter 3. This speeds up the proof checking of the Isabelle code by approximately a factor of 7, as the measurements in Section 3.3 show. The additional lemmata for the weak and the strength rule are illustrated in Figure 6.5. The base Isabelle theory file imports the additional lemmata in $SUM\_INTGERS\_complex\_implication\_lemmata$ together with the proof checker in its header. When feeding the proof checker with the Isabelle theory file, the additional lemmata will first be preprocessed. Their definition will then be added to the simplifier in the proof script such that all proof obligations can be proved: apply (simp add: sum\_i\_eq).
Figure 6.3: Bytecode proof generated by the PTC for the source proof in Figure 6.2.
theory SUM_INTEGERS
imports proof_checker SUM_INTEGERS_complex_implication_lemmata
begin

consts
  sum_integers_type:: TName
  sum_integers_INTERFACE_type:: TName

constdefs
  sumID:: MethodID
  "sumID ≡ 0"

constdefs
  sumPreCond:: Prec
  "sumPreCond ≡ ( λ s σ Z. (aInt(getV σ sum_n) > aInt(intV 1)) )"

constdefs
  sumPostCond:: Prec
  "sumPostCond ≡ ( λ s σ Z. (aInt(getV σ sum_Result) =
  aInt(intV((aInt(intV((aInt(getV σ sum_n)) * (aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1)))))))) div (aInt(intV 2)))))) )"

constdefs
  sumPostCondE:: Prec
  "sumPostCondE ≡ ( λ s σ Z. False )"

constdefs
  P001sumBody:: Prec
  "P001sumBody ≡ ( λ s σ Z. (aInt(getV σ sum_n) > aInt(intV 1)) )"

constdefs
  L001sumBody:: InstSpec
  "L001sumBody ≡ ( P001sumBody, 001, ( ldc (intV 0) ) )"

constdefs
  P002sumBody:: Prec
  "P002sumBody ≡ ( λ s σ Z. ((aInt(getV σ sum_n) > aInt(intV 1)) ∧
  (aInt(s[0]) = aInt(intV 0))) )"

constdefs
  L002sumBody:: InstSpec
  "L002sumBody ≡ ( P002sumBody, 002, ( stloc sum_Result ) )"

constdefs
  P003sumBody:: Prec
  "P003sumBody ≡ ( λ s σ Z. ((aInt(getV σ sum_n) > aInt(intV 1)) ∧
  (aInt(getV σ sum_Result) = aInt(intV 0))) )"

constdefs
  L003sumBody:: InstSpec
  "L003sumBody ≡ ( P003sumBody, 003, ( ldc (intV 0) ) )"
constdefs
P004sumBody:: Prec
"P004sumBody ≡ ( λ s σ Z. (((aInt(getV σ sum n) > aInt(intV 1)) ∧
(aInt(getV σ sum Result) = aInt(intV 0))) ∧ (aInt(s[0]) = aInt(intV 0))) )"

constdefs
L004sumBody:: InstSpec
"L004sumBody ≡ ( P004sumBody, 004, ( stloc sum i ) )"

constdefs
P005sumBody:: Prec
"P005sumBody ≡ ( λ σ Z. (aInt(getV σ sum n) > aInt(intV 1)) )"

constdefs
L005sumBody:: InstSpec
"L005sumBody ≡ ( P005sumBody, 005, ( ldc (intV 1) ) )"

constdefs
P006sumBody:: Prec
"P006sumBody ≡ ( λ σ Z. ((aInt(getV σ sum n) > aInt(intV 1)) ∧
((s[0]) = (intV 1))) )"

constdefs
L006sumBody:: InstSpec
"L006sumBody ≡ ( P006sumBody, 006, ( stloc sum Result ) )"

constdefs
P007sumBody:: Prec
"P007sumBody ≡ ( λ σ Z. (((aInt(getV σ sum n) > aInt(intV 1)) ∧
(aInt(getV σ sum Result) = aInt(intV 1))) ) ∧
(aInt(getV σ sum Result) = aInt(intV 1))) )" 

constdefs
L007sumBody:: InstSpec
"L007sumBody ≡ ( P007sumBody, 007, ( ldc (intV 2) ) )"

constdefs
P008sumBody:: Prec
"P008sumBody ≡ ( λ σ Z. (((aInt(getV σ sum n) > aInt(intV 1)) ∧
(aInt(getV σ sum Result) = aInt(intV 1))) ) ∧
(aInt(getV σ sum Result) = aInt(intV 1))) ) ∧
(aInt(getV σ sum i) = aInt(intV 2))) )"

constdefs
L008sumBody:: InstSpec
"L008sumBody ≡ ( P008sumBody, 008, ( stloc sum i ) )"

constdefs
P009sumBody:: Prec
"P009sumBody ≡ ( λ σ Z. (((aInt(getV σ sum n) > aInt(intV 1)) ∧
(aInt(getV σ sum Result) = aInt(intV 1))) ) ∧
(aInt(getV σ sum Result) = aInt(intV 1))) ) ∧
(aInt(getV σ sum i) = aInt(intV 2))) )"

constdefs
L009sumBody:: InstSpec
"L009sumBody ≡ ( P009sumBody, 009, ( nop ) )"
constdefs
P010sumBody:: Prec
"P010sumBody ≡ (λ s σ Z. (((aInt(getV σ sum_Result) = aInt(intV((aInt(intV((aInt(getV σ sum_i)) - (aInt(intV 1))))) * (aInt(getV σ sum_i)))) div (aInt(intV 2)))) ∨ (aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1))))) >= aInt(getV σ sum_i)) ∧ (aInt(getV σ sum_i) > aInt(intV 1))) )"

constdefs
L010sumBody:: InstSpec
"L010sumBody ≡ ( P010sumBody, 010, ( br 020 ) )"

constdefs
P011sumBody:: Prec
"P011sumBody ≡ (λ s σ Z. ((aInt(getV σ sum_i) = aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1))))) ∨ (((aInt(intV((aInt(getV σ sum_i)) + (aInt(intV 1)))) div (aInt(intV 2)))) ∨ (aInt(intV((aInt(getV σ sum_i)) + (aInt(intV 1))))))) div (aInt(intV 2)))) ∨ (aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1))))) >= aInt(getV σ sum_i)) ∧ (aInt(getV σ sum_i) > aInt(intV 1)) )"

constdefs
L011sumBody:: InstSpec
"L011sumBody ≡ ( P011sumBody, 011, ( nop ) )"

constdefs
P012sumBody:: Prec
"P012sumBody ≡ (λ s σ Z. ((aInt(getV σ sum_i) = aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1))))) ∨ (((aInt(intV((aInt(getV σ sum_i)) + (aInt(intV 1)))) div (aInt(intV 2)))) ∨ (aInt(intV((aInt(getV σ sum_i)) + (aInt(intV 1))))))) div (aInt(intV 2)))) ∨ (aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1))))) >= aInt(getV σ sum_i)) ∧ (aInt(getV σ sum_i) > aInt(intV 1)) )" ∧ (s 0) = (getV σ sum_Result)) )"

constdefs
L012sumBody:: InstSpec
"L012sumBody ≡ ( P012sumBody, 012, ( ldloc sum_Result ) )"

constdefs
P013sumBody:: Prec
"P013sumBody ≡ (λ s σ Z. ((aInt(getV σ sum_i) = aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1))))) ∨ (((aInt(intV((aInt(getV σ sum_i)) + (aInt(intV 1)))) div (aInt(intV 2)))) ∨ (aInt(intV((aInt(getV σ sum_i)) + (aInt(intV 1))))))) div (aInt(intV 2)))) ∨ (aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1))))) >= aInt(getV σ sum_i)) ∧ (aInt(getV σ sum_i) > aInt(intV 1)) )" ∧ (s[0]) = (getV σ sum_Result)) )"

constdefs
L013sumBody:: InstSpec
"L013sumBody ≡ ( P013sumBody, 013, ( ldloc sum_i ) )"
constdefs
P014sumBody:: Prec
  "P014sumBody ≡ ( λ s σ Z. (((aInt(getV[σ,sum_i]) ~=
      aInt(intV((aInt(getV[σ,sum_n]) + (aInt(intV 1))))) ∧
      ((aInt(intV((aInt(getV[σ,sum_Result]) + (aInt(getV[σ,sum_i]))) =
        aInt(intV((aInt(getV[σ,sum_i]) * (aInt(intV((aInt(getV[σ,sum_i]) + (aInt(intV 1)))))))) div (aInt(intV 2)))))) ∧
      (aInt(intV((aInt(getV[σ,sum_n]) + (aInt(intV 1)))) >= aInt(getV[σ,sum_i]))) ∧
      (aInt(getV[σ,sum_i]) > aInt(intV 1)))) ∧
      (intV((aInt(s[⌈1⌋]) + (aInt(s[⌈0⌋])))) = (intV((aInt(getV[σ,sum_Result]) + (aInt(getV[σ,sum_i]))) ))) )"

constdefs
L014sumBody:: InstSpec
  "L014sumBody ≡ ( P014sumBody, 014, ( iladd ) )"

constdefs
P015sumBody:: Prec
  "P015sumBody ≡ ( λ s σ Z. (((aInt(getV[σ,sum_i]) ~=
      aInt(intV((aInt(getV[σ,sum_n]) + (aInt(intV 1))))) ∧
      ((aInt(intV((aInt(getV[σ,sum_Result]) + (aInt(getV[σ,sum_i]))) =
        aInt(intV((aInt(getV[σ,sum_i]) * (aInt(intV((aInt(getV[σ,sum_i]) + (aInt(intV 1)))))))) div (aInt(intV 2)))))) ∧
      (aInt(intV((aInt(getV[σ,sum_n]) + (aInt(intV 1)))) >= aInt(getV[σ,sum_i]))) ∧
      (aInt(getV[σ,sum_i]) > aInt(intV 1)))) ∧
      ((s[⌈0⌋]) = (getV[σ,sum_i])) )"

constdefs
L015sumBody:: InstSpec
  "L015sumBody ≡ ( P015sumBody, 015, ( stloc sum[σ,sum_i] ) )"

constdefs
P016sumBody:: Prec
  "P016sumBody ≡ ( λ s σ Z. (((aInt(getV[σ,sum_i]) ~=
      aInt(intV((aInt(getV[σ,sum_n]) + (aInt(intV 1))))) ∧
      ((aInt(getV[σ,sum_Result]) = aInt(intV((aInt(intV((aInt(getV[σ,sum_i]) * (aInt(intV((aInt(getV[σ,sum_i]) + (aInt(intV 1)))))))) div (aInt(intV 2)))))) ∧
      (aInt(intV((aInt(getV[σ,sum_n]) + (aInt(intV 1)))) >= aInt(getV[σ,sum_i]))) ∧
      (aInt(getV[σ,sum_i]) > aInt(intV 1)))) ∧
      (s[⌈0⌋]) = (getV[σ,sum_i])) )"

constdefs
L016sumBody:: InstSpec
  "L016sumBody ≡ ( P016sumBody, 016, ( ldloc sum[σ,sum_i] ) )"

constdefs
P017sumBody:: Prec
  "P017sumBody ≡ ( λ s σ Z. (((aInt(getV[σ,sum_i]) ~=
      aInt(intV((aInt(getV[σ,sum_n]) + (aInt(intV 1))))) ∧
      ((aInt(getV[σ,sum_Result]) = aInt(intV((aInt(intV((aInt(getV[σ,sum_i]) * (aInt(intV((aInt(getV[σ,sum_i]) + (aInt(intV 1)))))))) div (aInt(intV 2)))))) ∧
      (aInt(intV((aInt(getV[σ,sum_n]) + (aInt(intV 1)))) >= aInt(getV[σ,sum_i]))) ∧
      (aInt(getV[σ,sum_i]) > aInt(intV 1)))) ∧
      (s[⌈0⌋]) = (getV[σ,sum_i])) )"

constdefs
L017sumBody:: InstSpec
  "L017sumBody ≡ ( P017sumBody, 017, ( ldc (intV 1) ) )"
constdefs
P018sumBody:: Prec
"P018sumBody ≡ (λ s σ Z. (((aInt(getV σ sum_i) == aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1)))))) ∧ (((aInt(getV σ sum_Result) = aInt(intV((aInt(getV σ sum_i)) * (aInt(intV((aInt(getV σ sum_i)) + (aInt(intV 1)))))) div (aInt(intV 2)))))) ∧ (aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1))))) >= aInt(getV σ sum_i)) ∧ (aInt(getV σ sum_i) > aInt(intV 1)))) ∧ ((intV((aInt(s⌈1⌉)) + (aInt(s⌈0⌉)))) = (intV((aInt(getV σ sum_i)) + (aInt(intV 1)))))) )" 

constdefs
L018sumBody:: InstSpec
"L018sumBody ≡ (P018sumBody, 018, (iladd ))"

constdefs
P019sumBody:: Prec
"P019sumBody ≡ (λ s σ Z. (((aInt(getV σ sum_i) == aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1)))))) ∧ (((aInt(getV σ sum_Result) = aInt(intV((aInt(intV((aInt(getV σ sum_i)) - (aInt(intV 1)))) * (aInt(getV σ sum_i))))) div (aInt(intV 2)))))) ∧ (aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1))))) >= aInt(getV σ sum_i)) ∧ (aInt(getV σ sum_i) > aInt(intV 1)))) ∧ ((aInt(s⌈0⌉)) = (aInt(getV σ sum_i) = aInt(intV((aInt(getV σ sum_i)) + (aInt(intV 1)))))) )" 

constdefs
L019sumBody:: InstSpec
"L019sumBody ≡ (P019sumBody, 019, (stloc sum_i ))"

constdefs
P020sumBody:: Prec
"P020sumBody ≡ (λ s σ Z. (((aInt(getV σ sum_Result) = aInt(intV((aInt(intV((aInt(intV((aInt(getV σ sum_i)) - (aInt(intV 1)))) * (aInt(getV σ sum_i))))) div (aInt(intV 2)))))) ∧ (aInt(intV((aInt(getV σ sum_i)) + (aInt(intV 1))))) >= aInt(getV σ sum_i)) ∧ (aInt(getV σ sum_i) > aInt(intV 1)))) ∧ (aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1))))) = (intV((aInt(getV σ sum_i)) + (aInt(intV 1)))))) )" 

constdefs
L020sumBody:: InstSpec
"L020sumBody ≡ (P020sumBody, 020, (ldloc sum_i ) )"

constdefs
P021sumBody:: Prec
"P021sumBody ≡ (λ s σ Z. (((aInt(getV σ sum_Result) = aInt(intV((aInt(intV((aInt(intV((aInt(getV σ sum_i)) - (aInt(intV 1)))) * (aInt(getV σ sum_i))))) div (aInt(intV 2)))))) ∧ (aInt(intV((aInt(getV σ sum_i)) + (aInt(intV 1))))) >= aInt(getV σ sum_i)) ∧ (aInt(getV σ sum_i) > aInt(intV 1)))) ∧ ((aInt(s⌈0⌉)) = aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1))))) = (intV((aInt(getV σ sum_i)) = aInt(intV((aInt(getV σ sum_i)) + (aInt(intV 1)))))) )" 

constdefs
L021sumBody:: InstSpec
"L021sumBody ≡ (P021sumBody, 021, (ldloc sum_n ) )"
constdefs
P022sumBody:: Prec
"P022sumBody ≡ (λ s σ Z. (((aInt(getV σ sum_Result) = aInt(intV((aInt(intV((aInt(intV((aInt(intV((aInt(getV σ sum_i)) - (aInt(intV 1)))) * (aInt(getV σ sum_i)))) div (aInt(intV 2))))))) ∧ (aInt((aInt(getV σ sum_n)) + (aInt(intV 1)))) >= aInt(getV σ sum_i)) ∧ (aInt(getV σ sum_i) > aInt(intV 1))) = (aInt((aInt(intV((aInt(intV((aInt(getV σ sum_i)) - (aInt(intV 1)))) * (aInt(getV σ sum_i)))) div (aInt(intV 2))))))) ∧ (aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1)))) >= aInt(getV σ sum_i)) ∧ (aInt(getV σ sum_i) > aInt(intV 1))) ∧ (aInt(s[1]) = aInt((aInt(s[0]) + (aInt(intV 1))))) = (aInt((aInt(intV((aInt(intV((aInt(getV σ sum_i)) - (aInt(intV 1)))) * (aInt(getV σ sum_i)))) div (aInt(intV 2))))))) ∧ (aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1)))))) )""

constdefs
L022sumBody:: InstSpec
"L022sumBody ≡ (P022sumBody, 022, (ldc (intV 1)))"

constdefs
P023sumBody:: Prec
"P023sumBody ≡ (λ s σ Z. (((aInt(getV σ sum_Result) = aInt(intV((aInt(intV((aInt(intV((aInt(intV((aInt(getV σ sum_i)) - (aInt(intV 1)))) * (aInt(getV σ sum_i)))) div (aInt(intV 2))))))) ∧ (aInt((aInt(getV σ sum_n)) + (aInt(intV 1)))) >= aInt(getV σ sum_i)) ∧ (aInt(getV σ sum_i) > aInt(intV 1))) = (aInt((aInt(intV((aInt(intV((aInt(getV σ sum_i)) - (aInt(intV 1)))) * (aInt(getV σ sum_i)))) div (aInt(intV 2))))))) ∧ (aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1)))) >= aInt(getV σ sum_i)) ∧ (aInt(getV σ sum_i) > aInt(intV 1))) ∧ (aInt(s[2]) = aInt((aInt(s[1]) + (aInt(s[0])))))) = (aInt((aInt(intV((aInt(intV((aInt(getV σ sum_i)) - (aInt(intV 1)))) * (aInt(getV σ sum_i)))) div (aInt(intV 2))))))) ∧ (aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1)))))) )""

constdefs
L023sumBody:: InstSpec
"L023sumBody ≡ (P023sumBody, 023, (iladd ))"

constdefs
P024sumBody:: Prec
"P024sumBody ≡ (λ s σ Z. (((aInt(getV σ sum_Result) = aInt(intV((aInt(intV((aInt(intV((aInt(intV((aInt(getV σ sum_i)) - (aInt(intV 1)))) * (aInt(getV σ sum_i)))) div (aInt(intV 2))))))) ∧ (aInt((aInt(getV σ sum_n)) + (aInt(intV 1)))) >= aInt(getV σ sum_i)) ∧ (aInt(getV σ sum_i) > aInt(intV 1))) = (aInt((aInt(intV((aInt(intV((aInt(getV σ sum_i)) - (aInt(intV 1)))) * (aInt(getV σ sum_i)))) div (aInt(intV 2))))))) ∧ (aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1)))) >= aInt(getV σ sum_i)) ∧ (aInt(getV σ sum_i) > aInt(intV 1))) ∧ (aInt(s[1]) = aInt((aInt(s[0]))) = (aInt((aInt(intV((aInt(intV((aInt(getV σ sum_i)) - (aInt(intV 1)))) * (aInt(getV σ sum_i)))) div (aInt(intV 2))))))) ∧ (aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1))))) )""

constdefs
L024sumBody:: InstSpec
"L024sumBody ≡ (P024sumBody, 024, (ilceq ))"

constdefs
P025sumBody:: Prec
"P025sumBody ≡ (λ s σ Z. (((aInt(getV σ sum_Result) = aInt(intV((aInt(intV((aInt(intV((aInt(intV((aInt(getV σ sum_i)) - (aInt(intV 1)))) * (aInt(getV σ sum_i)))) div (aInt(intV 2))))))) ∧ (aInt((aInt(getV σ sum_n)) + (aInt(intV 1)))) >= aInt(getV σ sum_i)) ∧ (aInt(getV σ sum_i) > aInt(intV 1))) = (aInt((aInt(intV((aInt(intV((aInt(getV σ sum_i)) - (aInt(intV 1)))) * (aInt(getV σ sum_i)))) div (aInt(intV 2))))))) ∧ (aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1)))) >= aInt(getV σ sum_i)) ∧ (aInt(getV σ sum_i) > aInt(intV 1))) ∧ (aInt(s[0]) = boolV((aInt((aInt(intV((aInt(intV((aInt(getV σ sum_i)) - (aInt(intV 1)))) * (aInt(getV σ sum_i)))) div (aInt(intV 2))))))) ∧ (aInt(intV((aInt(getV σ sum_n)) + (aInt(intV 1))))) )""

constdefs
L025sumBody:: InstSpec
"L025sumBody ≡ (P025sumBody, 025, (brfalse 011 ))"
constdefs
  sumBody:: CilProof
  "sumBody ≡ [L001sumBody, L002sumBody, L003sumBody, L004sumBody,
  L005sumBody, L006sumBody, L007sumBody, L008sumBody, L009sumBody,
  L010sumBody, L011sumBody, L012sumBody, L013sumBody, L014sumBody,
  L015sumBody, L016sumBody, L017sumBody, L018sumBody, L019sumBody,
  L020sumBody, L021sumBody, L022sumBody, L023sumBody, L024sumBody,
  L025sumBody]"
constdefs
  sumET:: ExcTable
  "sumET ≡ []"  
constdefs
  sum:: MethodDecl
  [simp]:"sum ≡ (sumID, sumPreCond, sumBody, sumET, (sumPostCond,
  sumPostCondE))"
constdefs
  sum_integers_body:: ClassBody
  [simp]:"sum_integers_body ≡ [sum]"
constdefs
  sum_integers:: ClassDeclaration
  [simp]:"sum_integers ≡ (refT sum_integers_type, sum_integers_body)"
constdefs
  myProgram:: CilProgram
  [simp]:"myProgram ≡ [sum_integers]"
defs
cil_subtypeSet_def[simp]: "cil_subtypeSet ≡ {(sum_integers_type, sum_integers_body)}"

theorem myProgram_theorem: "(is_program_safe (VCGen myProgram))"
  apply (simp)
  apply (unfold sumBody_def )
  apply (unfold VCGenMethod_base VCGenMethod_rec)
  apply (fold sumBody_def )
  apply (fold sum_def sum_integers_body_def )
  apply (fold sum_integers_def myProgram_def)
  apply (unfold L001sumBody_def L002sumBody_def L003sumBody_def
  L004sumBody_def L005sumBody_def L006sumBody_def L007sumBody_def
  L008sumBody_def L009sumBody_def L010sumBody_def L011sumBody_def
  L012sumBody_def L013sumBody_def L014sumBody_def L015sumBody_def
  L016sumBody_def L017sumBody_def L018sumBody_def L019sumBody_def
  L020sumBody_def L021sumBody_def L022sumBody_def L023sumBody_def
  L024sumBody_def L025sumBody_def )
  apply (unfold VCGenInst_def assn_imp_def)
apply (simp del: VCGenInstCaseAnalysis_def is_program_safe_base is_program_safe_rec)
apply (fold sum_def sum_integers_body_def )
apply (fold sum_integers_def myProgram_def)

apply (unfold VCGenInstCaseAnalysis_def)
apply (simp del: Wp_def is_program_safe_base is_program_safe_rec)

apply (unfold sumBody_def)

apply (simp del: Wp_def is_program_safe_base is_program_safe_rec)
apply (unfold Wp_def)
apply (unfold SuccLabel_base SuccLabel_rec getLabel_def getPre_def)
apply (simp del: is_program_safe_base is_program_safe_rec)


apply (simp del: is_program_safe_base is_program_safe_rec)
apply (simp add: sum_i_eq)
done
end

Figure 6.4: Generated Isabelle code for the bytecode proof in Figure 6.3
theory SUM_INTEGERS_complex_implication_lemmata
imports proof_checker
begin

constdefs
  sum_n:: VarName [simp]: "sum_n ≡ 5"

constdefs
  sum_Result:: VarName [simp]: "sum_Result ≡ 6"

constdefs
  sum_i:: VarName [simp]: "sum_i ≡ 7"

lemma sum_integers_method_sum_imp_lemma_0: "(((aInt(intV (aInt(getV σ sum_n)) > aInt(intV 1)) ∧ (aInt(getV σ sum_Result) = aInt(intV 1))) ∨ (aInt(getV σ sum_i) = aInt(intV 2))) → (((aInt(getV σ sum_Result) = aInt(intV (aInt((aInt(intV ((aInt(intV (aInt(getV σ sum_i)) - (aInt(intV 1))))) * (aInt(getV σ sum_i)))) div (aInt(intV 2))))) ∧ (aInt(intV ((aInt(getV σ sum_i)) + (aInt(intV 1))]))) == aInt(getV σ sum_i))) ∧ (aInt(intV (aInt(intV ((aInt(intV ((aInt(intV (sum_i)) * (aInt(intV ((aInt(intV (sum_i)) + (aInt(intV 1)))))))) div (aInt(intV 2))))))) ∧ (aInt(intV ((aInt(getV σ sum_i)) + (aInt(intV 1))))) >= aInt(getV σ sum_i))) ∧ (aInt(getV σ sum_i) > aInt(intV 1)))"
apply (auto)
done

lemma sum_i_eq: "(((i - (1::int)) * i) div 2) + i = i * (i + 1) div 2"
apply (simp add: ring_simps)
apply (arith)
done

lemma sum_integers_method_sum_imp_lemma_1: "(((aInt(intV (aInt(getV σ sum_i)) =~ aInt(intV (aInt(intV ((aInt(intV (aInt(getV σ sum_i)) + (aInt(intV 1))))))) ∧ (((aInt(getV σ sum_Result) = aInt(intV (aInt((aInt(intV ((aInt(intV ((aInt(intV (sum_i)) - (aInt(intV 1))))) * (aInt(getV σ sum_i)))) div (aInt(intV 2))))) ∧ (aInt(intV ((aInt(getV σ sum_i)) + (aInt(intV 1))))) == aInt(getV σ sum_i))) ∧ (aInt(intV (aInt(intV ((aInt(intV ((aInt(intV (sum_i)) * (aInt(intV ((aInt(intV (sum_i)) + (aInt(intV 1)))))))) div (aInt(intV 2))))))) ∧ (aInt(intV ((aInt(getV σ sum_i)) + (aInt(intV 1))))) >= aInt(getV σ sum_i))) ∧ (aInt(getV σ sum_i) > aInt(intV 1)))))
apply (simp add: sum_i_eq)
done

end

Figure 6.5: Additional Isabelle code, imported by the main Isabelle theory file from Figure 6.4, containing proofs for complex implication lemmata for the weak and strong proofs of the source proof in Figure 6.2.
Chapter 7

Conclusion and Future Work

Section 7.1 of this chapter concludes the thesis by summarizing the achievements of this thesis. Section 7.2 describes further optimization ideas and gives an outlook to future work.

7.1 Conclusion

The goal of this thesis was to embed Proof-Carrying Components, generated by the Proof-Transforming Compiler, into Isabelle.

We have implemented an Isabelle code generator which builds Isabelle code that is readable by the proof checker. If the Proof-Transforming Compiler produced a wrong specification or a wrong proof for a component, the proof checker rejects the component.

We have defined a formalization of the embedding, by specifying translation functions for the shallow embedding of the components contracts and the deep embedding of the components bytecode instructions. The formalization gives a concise specification of the translations to Isabelle without talking about implementation issues.

Furthermore, we developed a model for the Isabelle theory file to store the temporary generated code. Using the intermediate model gives much more flexibility to generate the final Isabelle code. The intermediate code can be postprocessed, such that generating the optimized proof script or changing the order of the Isabelle constant definition elements requires no additional visit of the bytecode tree and becomes a trivial task.

Optimizations. To speed up the proof checking, we have shown how to build an optimized proof script for proving the proof obligations produced by the verification condition generator. Using the optimized proof script, the proof checking is at least two times faster as when checking the proof by applying the simplifier solely.

Additionally, we have developed algorithms for simplifying bytecode proof expressions such as \( \{ \text{Result} := x + 1 \land y + 5 = y + 5 \} \) to \( \{ \text{Result} := x + 1 \} \) and \( \{ v > 0 \land (s(0) + v = \text{balance} + v) \} \) to \( \{ v > 0 \land (s(0) = \text{balance}) \} \). Applying
the simplifications to the bytecode proof produced by the Proof-Transforming Compilation improves the readability of the bytecode without changing its validity. The simplifications also speed up the proof checking of the generated Isabelle code since less contract expressions have to be checked.

**Extending the Proof-Transforming Compilation.** We extended the Proof-Transforming Compiler with the functionality to work with several classes. This makes it also possible to work with contract expressions that use different classes, such as type functions \texttt{is\_equal} or \texttt{conforms\_to}.

Besides that, we extended the PTC-EiffelStudio implementation with an interface for the Isabelle code generator, to be capable to embed the bytecode proof produced by the Proof-Transforming Compiler directly in EiffelStudio to Isabelle. The simple and informative interface allows a user to easily create the Isabelle code without needing to invoke an additional external program.

### 7.2 Future Work

The following features could be improved or added to the Proof-Transforming Compiler or the Isabelle embedding of the Proof-Carrying Component framework:

**Proof-Transforming Compilation**  We plan to investigate on how Proof-Transforming Compiler could be developed further, to be able to work with compiler optimizations.

Furthermore, we plan to extend the Proof-Transforming Compilers translation capabilities. It would be nice, if the the Proof-Transforming Compiler would also support the translation of agents.

A more general plan is to perform a case study which analyzes the feasibility of the usage of Proof-Transforming Compilation.

**Isabelle Code Generator**  The current implementation sometimes generates Isabelle type information for contract expressions that could be omitted, due to Isabelle's neat type inference capabilities. An additional visit of the generated contract expressions could simplify the vacuous type information.
Appendix A

Class Diagrams

Isabelle Generator  Figure A.1 shows all classes in the Isabelle Generator cluster. The \textit{isabelle} – \textit{theory} – \textit{model} subcluster is explained in the next paragraph. Figure A.2 gives an overview on the organization of the classes. The class \textit{ISABELLE GENERATOR} is the starting point for a new bytecode proof embedding. The class \textit{ISABELLE GENERATOR MAIN} is used for testing purposes and standalone usage.
Theory Model  Figure A.3 gives an overview on the classes that make up the Isabelle theory file model described in Section 4.2. Figure A.4 shows the BON diagram for the theory model. The class *ISABELLE_THEORY* is the root element of every Isabelle theory model.

![Figure A.3: Theory Model Cluster](image-url)
Figure A.4: BON Diagram for the Isabelle Theory File Model Classes
Appendix B

Isabelle Code

B.1 Proof Checker

theory proof_checker
imports AdditionalLemmas
begin

(* Datatype definitions for CIL programs (with proofs) *)

(* Instructions *)

types Label = nat
VarID = nat
MethodID = nat

datatype Instruction = ldloc VarName
| ldc Value
| stloc VarName
| iladd
| ilsub
| ilmul
| ildiv
| ilrem
| ilcmpeq
| ilcneq
| ilcgt
| ilclt
| ilcge
| ilcle
| iland
| ilor
| ilneg
| br Label
| brtrue Label
| bfalse Label
| nop
| ret
types ExcEntry = "Label × Label × Label × Type"
ExcTable = "ExcEntry list"

constdefs
is_handled:: "ExcEntry ⇒ Label ⇒ Type ⇒ bool"
[simp]: "(is_handled e l t) ≡ (case e of (st, end, targ, t2) ⇒
(if (st ≤ l ∧ l < end ∧ t ≤ t2) then True else False ))"

handlerEntry:: "ExcEntry ⇒ Label"
[simp]: "handlerEntry e ≡ (case e of (st, end, targ, t2) ⇒ targ )"

consts
handler:: "ExcTable ⇒ Label ⇒ Type ⇒ Label"

primrec
"handler [] l t = arbitrary "
"handler (x#xs) l t = (if (is_handled x l t) then (handlerEntry x) else
(handler xs l t))"

consts
is_caught :: "ExcTable ⇒ Label ⇒ Type ⇒ bool"

primrec
"is_caught [] l t = False "
"is_caught (x#xs) l t = (if (is_handled x l t) then True else (is_caught
xs l t))"

(* CIL proofs *)

types
Prec = "bool assn"
Precondition = Prec
Postcondition = "Prec × Prec"

types InstSpec = "Prec × Label × Instruction"
CilProof = "InstSpec list"

types MethodDecl = "MethodID × Precondition × CilProof ×
ExcTable × Postcondition"

ClassBody = "MethodDecl list"

ClassDeclaration = "Type × ClassBody"

CilProgram = "ClassDeclaration list"
(* Auxiliary functions for method invocation *)

constdefs
get_type:: "ClassDeclaration ⇒ Type"
      [simp]: "get_type c ≡ (case c of (t, cbody) ⇒ t)"

get_body:: "ClassDeclaration ⇒ ClassBody"
      [simp]: "get_body c ≡ (case c of (t, cbody) ⇒ cbody)"

get_id:: "MethodDecl ⇒ MethodID"
      [simp]: "get_id mdecl ≡ (case mdecl of (mid, pre, proof, et, post) ⇒ mid)"

get_precondition:: "MethodDecl ⇒ Prec"
      [simp]: "get_precondition mdecl ≡ (case mdecl of (mid, pre, proof, et, post) ⇒ pre)"

get_postcondition:: "MethodDecl ⇒ Prec"
      [simp]: "get_postcondition mdecl ≡ (case mdecl of (mid, pre, proof, et, post) ⇒ (case post of (postN, postE) ⇒ postN))"

consts
findClass:: "CilProgram ⇒ Type ⇒ ClassBody"
primrec
"findClass [] t = arbitrary"
"findClass (x#xs) t = (if (get_type x)=t then (get_body x) else (findClass xs t))"

consts
findMethod:: "ClassBody ⇒ MethodID ⇒ MethodDecl"
primrec
"findMethod [] m = arbitrary"
"findMethod (x#xs) m = (if (get_id x)=m then x else (findMethod xs m))"

constdefs
get_prec::"CilProgram ⇒ Type ⇒ MethodID ⇒ Prec"
      [simp]: "get_prec prog t m ≡ get_precondition(findMethod prog t m)"

get_post::"CilProgram ⇒ Type ⇒ MethodID ⇒ Prec"
      [simp]: "get_post prog t m ≡ get_postcondition(findMethod prog t m)"

(* Labels and successor *)

constdefs
getLabel:: "InstSpec ⇒ Label"
      [simp]: "getLabel i ≡ case i of (p, l, i) ⇒ l"

constdefs
getPre:: "InstSpec ⇒ Prec"
constdefs
Wp :: "Instruction ⇒ Prec ⇒ CilProof ⇒ Prec ⇒ Prec"[simp]: "Wp i sucPre proof post ≡
(case i of
ldloc v ⇒ (unshift (subst_x sucPre v))
| ldc n ⇒ (unshift (subst_c sucPre n))
| stloc v ⇒ (λ s σ Z.(subst_σ (shift sucPre) σ v) s σ Z)
| iladd ⇒ (λ s σ Z.(shift sucPre) (adds s)) s σ Z
| ilsub ⇒ (λ s σ Z.(shift sucPre) (subs s)) s σ Z
| imul ⇒ (λ s σ Z.(shift sucPre) (muls s)) s σ Z
| idiv ⇒ (λ s σ Z.(shift sucPre) (divs s)) s σ Z
| irem ⇒ (λ s σ Z.(shift sucPre) (rems s)) s σ Z
| ilcneq ⇒ (λ s σ Z.(shift sucPre) (neqs s)) s σ Z
| ilceq ⇒ (λ s σ Z.(shift sucPre) (eqs s)) s σ Z
| ilcgt ⇒ (λ s σ Z.(shift sucPre) (cgts s)) s σ Z
| ilcgte ⇒ (λ s σ Z.(shift sucPre) (gtes s)) s σ Z
| ilclt ⇒ (λ s σ Z.(shift sucPre) (clts s)) s σ Z
| ilclte ⇒ (λ s σ Z.(shift sucPre) (ltes s)) s σ Z
| iland ⇒ (λ s σ Z.(shift sucPre) (ands s)) s σ Z
| ilor ⇒ (λ s σ Z.(shift sucPre) (ors s)) s σ Z
| ilneg ⇒ (λ s σ Z.(shift sucPre) (negIs s)) s σ Z
| br l2 ⇒ (λ s σ Z.(SuccLabel proof l2)) s σ Z
| brtrue l2 ⇒ (λ s σ Z.(assn_andi (assn_implication (λ s σ Z. s[0] = (boolV False)) (shift sucPre)) (assn_implication (λ s σ Z. s[0] = (boolV True)) (shift (SucLabel proof l2)))) s σ Z)
| brfalse l2 ⇒ (λ s σ Z.(assn_andi (assn_implication (λ s σ Z. (s[0]) = (boolV True))) (shift sucPre)) (assn_implication (λ s σ Z. (s[0]) = (boolV False))) (shift (SucLabel proof l2)))) s σ Z
| nop ⇒ sucPre
| ret ⇒ post
| ld fld fId ⇒ (λ s σ Z.(subst_c sucPre (getFV σ (getSi s 0) fId)) s σ Z)
| st fld fId ⇒ (λ s σ Z.(subst_st (shift sucPre) ((fst σ), (updtOS (σ[os]) iLoc (getSi s 1) fId) (s[0]))) s ((fst σ), (updtOS (σ[os]) (iLoc (getSi s 1) fId) (s[0]))) Z)
| newobj t ⇒ (λ s σ Z. (unshift (subst_c (shift sucPre (iAlloc σ t)) (oVal (newINS t (σ[os]))) s (iAlloc σ t) Z))
\[ \text{castc } t \Rightarrow (\lambda s \sigma Z. (\text{assn_andi succPre} (\lambda s \sigma Z. (\tau (s[0])) \preceq \text{refT} (t))) s \sigma Z) \]
\[ \text{callvirt } t m e \Rightarrow \text{arbitrary} \]
\[ \text{throw} \Rightarrow \text{arbitrary} \]

(* VCGen functions *)

**types**

VC = "bool"
ProofObligation = "VC list"

**constdefs**

PO_call:: "CilProgram \Rightarrow Type \Rightarrow MethodID \Rightarrow Value \Rightarrow Prec \Rightarrow Prec \Rightarrow bool"  
[simp]:"PO_call prog t m e pre sucPre \equiv 
(\text{assn_imp pre} (\lambda s \sigma Z. (\text{subst_}\sigma (\text{sub}_{\sigma 1} (\text{sub}_{\sigma} (\text{get}_{\sigma \text{prec}} prog t m) \sigma \text{PARAM}) \sigma \text{THIS}) s \sigma Z)) \land (\text{assn_imp} (\lambda s \sigma Z. (\text{subst}_{\sigma} (\text{sub}_{\sigma} (\text{get}_{\sigma \text{post}} prog t m) \sigma e) \sigma \text{RESULT}) s \sigma Z)) sucPre )  "

**constdefs**

handlerPost:: "ExcTable \Rightarrow Label \Rightarrow Type \Rightarrow Prec \Rightarrow CilProof \Rightarrow Prec"
[simp]:"handlerPost et l t post proof \equiv 
(if (is_caught et l t) then (SuccLabel proof (handler et l t)) else post)"

**constdefs**

PO_throw:: "ExcTable \Rightarrow Label \Rightarrow Prec \Rightarrow CilProof \Rightarrow Prec"
[simp]:"(PO_throw et l post proof) \equiv 
(\lambda s \sigma Z. (\text{handlerPost} et l (\tau (s[0])) post proof) s \sigma Z)  "

**constdefs**

VCGenInstFull:: "InstSpec \Rightarrow ExcTable \Rightarrow CilProof \Rightarrow CilProgram \Rightarrow Prec \Rightarrow Prec \Rightarrow VC"  
[simp]:"(VCGenInstFull inSp et xs prog postN postE) \equiv 
\begin{cases} 
\text{case inSp of} \ (p, l, i) \Rightarrow (\text{case i of} 
callvirt t m e \Rightarrow ( \text{PO_call prog t m e p (SuccL (1{l+1}) xs postN) }) 
\text{\text{\_ \Rightarrow (assn_imp p (PO_throw et l postE xs)})})
\end{cases}"

**constdefs**

VCGenInstCaseAnalysis:: "Prec \Rightarrow Label \Rightarrow Instruction \Rightarrow ExcTable \Rightarrow CilProof 
\Rightarrow CilProgram \Rightarrow Prec \Rightarrow Prec \Rightarrow VC"  
[simp]:"(VCGenInstCaseAnalysis p l i et xs prog postN postE) \equiv 
\begin{cases} 
\text{case i of} 
callvirt t m e \Rightarrow ( \text{PO_call prog t m e p (SuccL (1{l+1}) xs postN) }) 
\text{\_ \Rightarrow (assn_imp p (PO_throw et l postE xs)})
\end{cases}"

**constdefs**

VCGenInst:: "InstSpec \Rightarrow ExcTable \Rightarrow CilProof \Rightarrow CilProgram \Rightarrow Prec \Rightarrow Prec \Rightarrow VC"  
[simp]:"(VCGenInst inSp et xs prog postN postE) \equiv 
\begin{cases} 
\text{case inSp of} \ (p, l, i) \Rightarrow ( \text{VCGenInstCaseAnalysis p l i et xs prog postN postE})
\end{cases}"
consts
VCGenMethod:: "CilProof ⇒ ExcTable ⇒ CilProof ⇒ CilProgram ⇒ Prec ⇒ Prec ⇒ ProofObligation"
primrec
VCGenMethod_base: "VCGenMethod [] et ys prog postN postE = ( [True])"
VCGenMethod_rec: "VCGenMethod (x#xs) et ys prog postN postE = ( ((VCGenInst x et ys prog postN postE)# (VCGenMethod xs et ys prog postN postE)))"

c consts
VCGenBody:: "ClassBody ⇒ CilProgram ⇒ ProofObligation"
primrec
VCGenBody_base: "VCGenBody[] prog = ([True])"
VCGenBody_rec: "VCGenBody (x#xs) prog = (case x of (m, pre, cilP, excTable, post ) ⇒ (case post of (postN, postE) ⇒ ((VCGenMethod cilP excTable cilP prog postN postE) @ (VCGenBody xs prog))))"

c consts
VCGen2:: "CilProgram ⇒ CilProgram ⇒ ProofObligation"
primrec
VCGen2_base: "VCGen2 [] prog = [True]"
VCGen2_rec: "VCGen2 (x#xs) prog = (case x of (t, b) ⇒ ( (VCGenBody b prog) @ (VCGen2 xs prog)))"
ung
constdefs
VCGen:: "CilProgram ⇒ ProofObligation"
[simp]:"VCGen c ≡ VCGen2 c c"
c ons
is_program_safe:: "ProofObligation ⇒ bool"
primrec
is_program_safe_base: "is_program_safe [] = True"
is_program_safe_rec: "is_program_safe (x#xs) = (if x then (is_program_safe xs) else False )"
end
B.2 Example List

theory ExampleList
imports Datatype
begin

datatype 'a list = Nil ([])
  | Cons 'a ''a list" (infixr "#" 65)

primrec app :: ''a list ⇒ ''a list ⇒ ''a list" (infixr "@" 65)
where
  "[] @ ys = ys" |
  "(x # xs) @ ys = x # (xs @ ys)"

primrec rev :: ''a list ⇒ ''a list"
where
  "rev [] = []" |
  "rev (x # xs) = (rev xs) @ (x # [])"
Bibliography


