Proofs and Proof Transformations for Object-Oriented Programs

Martin Nordio

2009
PROOFS AND PROOF TRANSFORMATIONS FOR OBJECT-ORIENTED PROGRAMS

DISSERTATION

Submitted to

ETH ZURICH

for the degree of

DOCTOR OF SCIENCES

by

DARÍO MARTÍN NORDIO

M.Sc. CS, Universidad de la República, Uruguay

born
March 30th, 1979

citizen of
Argentina

accepted on the recommendation of

Prof. Dr. Bertrand Meyer, examiner
Prof. Dr. Gilles Barthe, co-examiner
Prof. Dr. Peter Müller, co-examiner
Prof. Dr. Jim Woodcock, co-examiner

2009
To Luciana
for her company and love.
ACKNOWLEDGEMENTS

I have arrived at the destination of my voyage, although, it may not be the final station. In spite of the traveling facilities of the twenty first century having improved compared to the ones in Phileas Fogg’s times, this journey has been very exciting for me.

The preparations for the journey started around mid-2003. After finishing my basic training\footnote{Licenciado in computer science}, I started to prepare my backpack. While waiting for my boat to depart, Jorge Aguirre suggested that I start a shorter trip\footnote{Master in computer science} to Uruguay. Jorge was my guide in the rough waters of Río de la Plata, and I came back to Argentina in 2005.

My voyage started in the port of Córdoba on a cold day in June 2005; the sky was cloudy and dark. The boat left on time for Madrid, where my cousin Pablo was waiting for me. After visiting the Santiago Bernabéu stadium and eating tapas and jamón serrano, I took a train to Zurich, where my guides were waiting for me.

Such an important voyage cannot be done without a guide who not only recommends, during a cold and dark night, to turn left or to turn right, but also a guide who shows you the uniqueness of the surroundings. The guides of my voyage have been Bertrand Meyer and Peter Müller.

Around the middle of the trip, I was riding my dapple grey horse in the Úetliberg hill. This hill is not high, but it is a very foggy area. I saw a tall man running; he was wearing a genoa t-shirt. He told me he was training for the London marathon. His name is Cristiano Calcagno.

One of the most interesting part of my trip was meeting other travelers, and sharing experiences. The expedition was enriching thanks to the company of my fellow SE and PM travelers\footnote{Past and present members of the chair of Software Engineering and Chair of Programming Methodology at ETH Zurich}: Ilinca Ciupa, Ádám Darvas, Werner Dietl, Pietro Ferrara, Carlo Furia, Nicu Georgian Fruja, Claudia Günthart, Xiaohui (Jenny) Jin, Ioannis Kassios, Hermann Lehner, Andreas Leitner, Sebastian Nanz, Piotr Nienaltowski, Benjamin Morandi,
I have met other travelers who were heading to different destinations, still I had the opportunity to share part of the trip with them; they are Michel Guex, Bruno Hauser, Manuel Hess, and Hasan Karahan.

One day in the cold winter of Zürich, I found a small flyer about an outsourced expedition called DOSE. The expedition consisted of discovering different cultures; in particular, I visited Russia, Ukraine, Italy, India, and Vietnam. I travelled with Peter Kolb, Roman Mitin, and Bertrand Meyer.

My family and my argentinean friends could not board the boat with me, however, they accompanied me from afar. Without their understanding and support, this trip could not have been done. A special thanks goes to my mother Lilian, Germán, Mariela, and my niece Sofía, my sister Luciana, my aunt Lili, the Espamer family, and Pablo and Mariana. My life fellow traveller Luciana was my company for the whole voyage; I am thankful for her company and love.

Today, my trip finishes: the day is sunny and the sky is blue. The results of this trip are described in the next pages, and it is known as PhD thesis.

Martín, Zürich 2009
CONTENTS

I  Overview 3

1  Carrying Proofs from Source to Target 5

2  Overview and Main Results 9

2.1  A Verification Process based on Proof Transforming Compilation . 9

2.2  Contributions . 11

2.2.1  Proofs for Object-Oriented Programs . 12

2.2.2  Proof-Transformations for Object-Oriented Programs . 12

II  Proofs 15

3  A Sound and Complete Logic for Object-Oriented Programs 17

3.1  The Core Language and its Semantics . 18

3.1.1  The Mate Language . 18

3.1.2  The Memory Model . 19

3.1.3  Operational Semantics . 22

3.1.4  A Programming Logic for Mate . 27

3.1.5  Example . 32

3.2  A Logic for Eiffel . 36

3.2.1  The Eiffel Language . 36

3.2.2  Operational Semantics . 36

3.2.3  A Programming Logic for Eiffel . 44

3.2.4  Example . 46

3.3  A Logic for Java-like Programs . 48

3.3.1  The Java Language . 48

3.3.2  Operational Semantics . 48

3.3.3  A Programming Logic for Java . 53

3.3.4  Example . 55

3.4  Soundness and Completeness . 57
### 3.5 Related Work

58

### 3.6 Lessons Learned

59

### 4 Reasoning about Function Objects

63

#### 4.1 The Agent Mechanism

64

#### 4.2 Agent Examples and their Verification Challenge

66

- **4.2.1** Formatter

- **4.2.2** Multi-Level Undo-Redo

- **4.2.3** Archive Example

67

68

#### 4.3 Verification Methodology

69

- **4.3.1** Specifying Function Objects

- **4.3.2** Axiomatic Semantics

69

71

#### 4.4 Automatic Proofs

73

- **4.4.1** Reasoning about agents with open arguments

- **4.4.2** Reasoning about Closed Arguments

- **4.4.3** Framing

- **4.4.4** Framing for Agents with Closed Arguments

- **4.4.5** Applications

- **4.4.6** Experiments

73

75

77

79

81

86

87

#### 4.5 Related Work

87

### III Proof Transformations

89

### 5 The CIL Language and its Logic

91

#### 5.1 The CIL Bytecode Language

91

#### 5.2 A Bytecode Logic for CIL

93

- **5.2.1** Method and Instruction Specifications

- **5.2.2** Rules

- **5.2.3** Exception Handling

- **5.2.4** Example

93

95

97

103

### 6 Proof-Transforming Compilation for the Core Language

107

#### 6.1 Translation Basics

107

#### 6.2 Proof Translation of Routines

108

- **6.2.1** Class Rule

- **6.2.2** Subtype Rule

- **6.2.3** Routine Implementation Rule

- **6.2.4** Language-Independent Rules for Routines

109

109

110

111

#### 6.3 Proof Translation of Expressions

115

- **6.3.1** Constants

116
# CONTENTS

6.3.2 Variables ......................................................... 116
6.3.3 Binary Expressions ............................................... 116
6.3.4 Unary Expressions ............................................... 117

6.4 Proof Translation of Instructions .................................. 117
  6.4.1 Assignment Axiom .............................................. 118
  6.4.2 Compound Rule ............................................... 119
  6.4.3 Conditional Rule .............................................. 119
  6.4.4 Check Axiom .................................................. 121
  6.4.5 Loop Rule .................................................... 121
  6.4.6 Read Attribute Axiom .......................................... 122
  6.4.7 Write Attribute Axiom ......................................... 123
  6.4.8 Routine Invocation Rule ....................................... 124
  6.4.9 Local Rule .................................................... 125
  6.4.10 Creation Rule ............................................... 127

6.5 Proof Translation of Language-Independent Rules .................. 127
  6.5.1 Strength Rule ................................................ 128
  6.5.2 Weak Rule ................................................... 128
  6.5.3 Conjunction/Disjunction Rules ............................... 129
  6.5.4 Invariant Rule ............................................... 130
  6.5.5 Substitution Rule ............................................ 131
  6.5.6 All rule and Ex rule ......................................... 131

6.6 Applications .................................................. 132

6.7 Soundness Theorems ............................................... 134

7 Eiffel-Specific Aspects of Proof-Transforming Compilation .......... 137
  7.1 Contract Translator .............................................. 138
    7.1.1 Translation Basics .......................................... 138
    7.1.2 Datatype Definitions ....................................... 139
    7.1.3 Mapping Eiffel Types to CIL ................................ 140
    7.1.4 Translation Functions ....................................... 141
    7.1.5 Example Translation ......................................... 144
  7.2 Proof Translator ................................................ 145
    7.2.1 Transformation Function Basics ............................. 146
    7.2.2 Proof Translation of Eiffel Expressions .................. 146
    7.2.3 Proof Translation of Instructions .......................... 148
  7.3 Applications .................................................. 152
  7.4 Soundness of the Contact Translator ............................. 154
# CONTENTS

8 Java-Specific Aspects of Proof-Transforming Compilation 155

8.1 Proof Translation using CIL ................................................. 156
  8.1.1 Translation Basics ................................................. 156
  8.1.2 Compound ............................................................. 156
  8.1.3 While Rule ............................................................ 157
  8.1.4 Break Rule ............................................................ 158
  8.1.5 Throw Rule ............................................................ 158
  8.1.6 Try-catch Rule ...................................................... 159
  8.1.7 Try-finally Rule .................................................... 160
  8.1.8 Soundness Theorem ................................................. 161

8.2 Proof Translation using Java Bytecode ................................. 162
  8.2.1 Translation Basics ................................................. 163
  8.2.2 Compound ............................................................. 164
  8.2.3 While Rule ............................................................ 165
  8.2.4 Try-Finally Rule .................................................... 165
  8.2.5 Break Rule ............................................................ 167
  8.2.6 Soundness Theorem ................................................. 169

8.3 Applications ............................................................... 170

8.4 Related Work ............................................................... 172

9 Implementation of the Proof-Transforming Compiler Scheme 179

9.1 A Proof-Transforming Compiler ........................................... 179

9.2 Proof Checker .............................................................. 181
  9.2.1 Instructions .......................................................... 182
  9.2.2 Exception Tables .................................................... 182
  9.2.3 CIL Proofs ............................................................ 183
  9.2.4 Weakest Precondition Function .................................. 184
  9.2.5 Verification Condition Generator ................................. 185
  9.2.6 Checking the CIL Proofs .......................................... 187

9.3 Experiments ................................................................. 188

10 Conclusions and Future Work 191

10.1 Conclusions ............................................................... 191

10.2 Future Work .............................................................. 192
  10.2.1 Proofs ............................................................... 192
  10.2.2 Proof-Transforming Compilation ................................. 193

A Notation 195
B Soundness and Completeness Proof of the Logic 197

B.1 Definitions and Theorems ........................................... 198
B.2 Soundness Proof ....................................................... 200
   B.2.1 Assignment Axiom ............................................. 200
   B.2.2 Compound Rule .................................................. 201
   B.2.3 Conditional Rule ............................................... 203
   B.2.4 Check Axiom .................................................... 204
   B.2.5 Loop Rule ....................................................... 205
   B.2.6 Read Attribute Axiom ......................................... 206
   B.2.7 Write Attribute Axiom ........................................ 207
   B.2.8 Local Rule ....................................................... 208
   B.2.9 Creation Rule ................................................... 209
   B.2.10 Rescue Rule .................................................... 210
   B.2.11 Once Functions Rule ......................................... 213
   B.2.12 Routine Implementation Rule ................................. 217
   B.2.13 Routine Invocation Rule ..................................... 218
   B.2.14 Class Rule ..................................................... 219
   B.2.15 Subtype Rule ................................................... 220
   B.2.16 Language-Independent Rules ................................. 221
B.3 Completeness Proof ................................................... 224
   B.3.1 Assignment Axiom ............................................. 225
   B.3.2 Compound Rule .................................................. 226
   B.3.3 Conditional Rule ............................................... 226
   B.3.4 Check Axiom .................................................... 227
   B.3.5 Loop Rule ....................................................... 227
   B.3.6 Read Attribute Axiom ......................................... 228
   B.3.7 Write Attribute Axiom ........................................ 228
   B.3.8 Local Rule ....................................................... 229
   B.3.9 Rescue Rule .................................................... 230
   B.3.10 Routine Implementation Rule ................................. 232
   B.3.11 Routine Invocation Rule ..................................... 232
   B.3.12 Virtual Routines ................................................ 235

C Soundness Proof of the Eiffel Proof-Transforming Compiler 237

C.1 Theorems ............................................................... 237
C.2 Soundness Proof of the Routine Translator ....................... 238
   C.2.1 Class Rule ..................................................... 238
   C.2.2 Conjunction Rule ............................................... 240
C.3 Soundness Proof of the Expression Translator .................... 240
   C.3.1 Constants ...................................................... 241
   C.3.2 Variables ...................................................... 242
Abstract

In modern development schemes the processing of programs often involves an intermediate step of translation to some intermediate code, often called “bytecode”. This intermediate code is executed through interpretation or a second phase of compilation known as “just-in-time”. The execution of bytecode programs can produce unexpected behavior, which may compromise security and correctness of a software system. An important issue to address is the verification of these programs also known as mobile code.

Expanding on the ideas of Proof-Carrying Code (PCC), we have introduced a verification process for bytecode programs based on proof-transforming compilation. The approach consists of translating proofs of object-oriented programs to bytecode proofs. The verification process is performed at the level of the source program making interaction easier than verifying bytecode programs. Then, a proof-transforming compiler translates automatically a contract-equipped program and its proof into bytecode representing both the program and the proof. Our approach addresses not only type safety properties, as in the original PCC work, but full functional correctness as expressed by the original contracts.

This thesis develops the foundations of proof-transforming compilation for object-oriented programs. The main results are: (1) operational and axiomatic semantics for a subset of C#, Eiffel, and Java; (2) a verification methodology for function objects; and (3) proof-transforming compilers for these languages. This thesis shows that certificates for bytecode programs can be generated automatically from certificates of object-oriented programs. The implemented prototype suggests that proof-transforming compilers can be applied to real programming languages.
Abstract

Moderni metodi di sviluppo software utilizzano un processo di traduzione da codice sorgente a codice di livello intermedio, detto “bytecode”. Questo codice viene eseguito da un interprete, o gestito in una seconda fase di compilazione detta “jitting”. L’esecuzione di bytecode può produrre risultati incorretti, che potrebbero compromettere la sicurezza e la correttezza dei sistemi di software. Un aspetto importante da considerare è anche la presenza di codice mobile.

Partendo dall’idea di Proof-Carrying Code (PCC), abbiamo proposto un’estensione del processo di verifica per bytecode basata sull’idea di proof-transforming compilation. L’approccio consiste nella traduzione di dimostrazioni di programmi object oriented. Il processo di verifica avviene automaticamente a partire da un programma provvisto di dimostrazione, e produce un nuovo programma annotato con una dimostrazione. L’approccio proposto si applica non solo a proprietà di tipo, come nel PCC originale, ma è in grado di esprimere correttezza funzionale.

Questa tesi sviluppa gli aspetti fondamentali di proof-transforming compilation per linguaggi object-oriented. I risultati principali sono: (1) semantica operazionale e assiomatica per un sottoinsieme di C#, Eiffel e Java; (2) metodologia di verifica per oggetti funzionali; (3) compilatori proof-transforming per questi linguaggi. Questa tesi dimostra che certificati per programmi bytecode possono essere generati automaticamente a partire da programmi object-oriented. L’implementazione di un prototipo suggerisce che i compilatori proof-transforming possono essere applicati a linguaggi di programmazione in uso corrente.
Abstract
Part I

Overview
Chapter 1

Carrying Proofs from Source to Target

The problem of software verification, hard enough in a traditional context, takes on new twists as advances in computing, designed to bring convenience and flexibility to users, also bring further headaches to verifiers. The problem arises because of the increased sophistication of our computing architectures. Along with new modes of computing originating from the role of the Internet, new modes of software deployment have emerged. Once we have written a program in a high-level language, instead of compiling it once and for all into machine code for execution on a given machine, we may generate intermediate code, often called “bytecode” (CIL on .NET, or JVM bytecode) and distribute it to numerous users who will execute it through either interpretation or a second phase of compilation known as “jitting”. What then can and should we verify?

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

The problem now becomes to derive from a proof of the source code a guarantee of correctness of the generated bytecode. Unlike the interpreter or jitter, the compiler is often outside of the operating system; even if it is trusted, there is no guarantee against a third party tampering with the intermediate code.

The notion of Proof-Carrying Code (PCC) was developed to address this issue; with PCC, a program producer develops code together with a formal proof (a certificate)
that it possesses certain desirable properties. The program consumer checks the proof before executing the code or, in the above scheme, before interpreting or jitting it. To achieve full automation, such work has mostly so far addressed a subset of program properties—those associated with security. In this work, we are interesting in proofs of full functional correctness, which typically require interactive verification and are beyond the capabilities of today’s certifying compilers.

Outline

This thesis addresses the problem of verifying functional correctness properties for bytecode programs. In Chapter 2, we describe our approach of proof-transforming compilation, and we briefly outline the main results and contributions.

In the second part, proofs, Chapter 3 presents a sound and complete logic for object-oriented programs. The chapter defines an operational and axiomatic semantics for a common subset of C#, Eiffel, and Java. Then, we extend the language to a subset of Eiffel that handles exception handling, once routines, and multiple inheritance. Finally, we develop a semantics for a subset of Java that handles abrupt termination using break, throw, try-catch, and try-finally instructions.

Chapter 4 develops a specification and verification methodology for function objects. We illustrate the methodology using Eiffel agents. The chapter first gives a brief introduction to the Eiffel agent mechanism, and then it shows some challenging examples. The chapter concludes with the verification methodology, applications, and related work.

In the third part, proof-transformations, Chapter 5 describes the bytecode language, CIL, and summarizes the bytecode logic developed by Bannwart and Müller [6]. This chapter also extends Bannwart and Müller’s logic with the CIL instructions try-catch and try-finally.

The proof-transforming compilation technique is presented in three chapters. Chapter 6 develops a proof-transforming compiler for a common subset of C#, Eiffel, and Java. This chapter presents: (1) the translation of virtual routine and routine implementation rules, (2) the translation of expressions, (3) the translation of basic instructions such as assignment and compound, and (4) the translation of language-independent rules. The chapter concludes with an example of the application of proof-transforming compilers.

Chapter 7 describes the Eiffel proof-transforming compiler. The compiler takes a proof in a Hoare-style logic for Eiffel, and produces a proof for CIL. This compiler consists of a contract translator, and an instruction translator. The chapter concludes with an example of the application of Eiffel PTC.

Chapter 8 develops the Java proof-transforming compiler. The Proof-Transforming Compiler (PTC) handles abrupt termination using break, throw, try-catch, and try-finally instructions. This chapter shows the translation using both CIL and JVM bytecode. The chapter concludes with a discussion of related work. Chapter 9 describes
the implementation of the proof-transforming compiler schema. Conclusions and future work are presented in Chapter 10.

Finally, Appendix A shows the notation used in the thesis; Appendix B proves soundness and completeness of the Eiffel logic; and Appendices C and D prove soundness of the Eiffel and Java PTC, respectively.
Carrying Proofs from Source to Target
Chapter 2

Overview and Main Results

The work reported in this thesis addresses the problem of verifying mobile code and other programs deployed through intermediate formats such as bytecode. We tackle the entire issue of functional correctness by introducing a proof-transforming compiler (PTC).

2.1 A Verification Process based on Proof Transforming Compilation

A process for building verified software with the help of the concepts developed in this thesis would involve the following steps:

- Verify the source program, taking advantage of proof technology at the programming language level. This step can involve interaction with the programmer or verification expert.

- Translate both the source program and the proof into intermediate code, using the proof-transforming compiler. This step is automatic.

- Before a user runs the code (through interpretation or jitting), check the proof. This checking is again an automatic task; it can be performed by a simple proof-checking tool.

A traditional compiler has as input a program in source form and, in the kind of architectures considered here, produces bytecode as output. With a proof-transforming compiler, the source program must be equipped with verification conditions to enable a proof of correctness; the input of the proof-transforming compiler includes, along with this source program, a proof of its correctness; and the output generates, along with the generated bytecode, the corresponding proof of its correctness. Figure 2.1 shows the
Overview and Main Results

Fig. 2.1: General Architecture of Proof-Transforming Compilation.

architecture of this approach. The code producer develops a program and a proof of the source program using a prover. Then, the PTC translates the proof producing the bytecode and its proof, which are sent to the code consumer. The proof checker verifies the proof of the bytecode program. If the bytecode proof is incorrect, the proof checker rejects the code; otherwise the bytecode program is executed. Similar to PCC, if a third party modifies the bytecode or the bytecode proof making the proof invalid, the proof checker would detect it, and the code would be rejected.

An important property of proof-transforming compilers is that they are not part of the trusted computing base of the Proof-Carrying Code infrastructure. If the compiler produces a wrong specification or a wrong proof for a component, the proof checker will reject the component. This approach combines the strengths of certifying compilers and interactive verification.
2.2 Contributions

Our approach inherits the advantages of PCC: (1) the process of checking the proof is fast and automatic; (2) there is no loss of performance in the bytecode program since first, the proof is checked, and then the code is executed (without inserting any run-time checks); (3) the overhead of developing the proof is done once and for all by the code producer; (4) the code consumer does not need to trust the code producer.

The main advantage of the verification process based on proof-transforming compilation against Proof-Carrying Code is that our approach addresses functional correctness. The approach supports heterogeneity of source programming languages and verification techniques for the proofs. Proofs can be developed for programs written in different programming languages such as C#, Eiffel, or Java, and then translated to a uniform format.

To cover a wide range of features of object-oriented languages, we have applied our approach to subsets of C#, Eiffel, and Java. These programming languages have many interesting object-oriented features such as two different exception handling mechanisms, single and multiple inheritance, and abrupt termination (for example using `break` instructions). We have studied these languages from two different point of views: the semantics and the proof transformation point of view. The first one allows us to analyze and to compare the different features such as the Eiffel and the Java exception handling mechanisms. The second one allows us to study how proofs can be translated for different programming languages. The proof translation for C# and Java is interesting because it shows how abrupt termination can be mapped to bytecode. The proof translation from Eiffel to CIL is interesting because of the difference in the support for inheritance; while Eiffel handles multiple inheritance, CIL supports single inheritance.

The novel concept of proof-transforming compilation has also attracted the interest of other researchers [15, 103, 114]. The main difference of the approach developed in this thesis and other approaches is how the proof-transforming compiler works. Previous work [15, 103, 114] has shown that proof obligations are preserved by compilation; the aim is to prove the equivalence between the verification conditions generated over the source code and the bytecode. Our approach demonstrates how proofs of programs using Hoare-style logic can be translated to proofs in bytecode. An advantage of our approach against the preservation of proof obligation (PPO) approach is that the trusted computing base of our approach consists only of a proof checker; in the PPO approach, the trusted computing base consists of a verification condition generator for the bytecode and a proof checker. For a detailed comparison of these approaches see Section 8.4.

2.2 Contributions

The contributions of this thesis can be classified into proofs for object-oriented programs and proof-transformations for object-oriented programs.
2.2.1 Proofs for Object-Oriented Programs

Semantics for C#, Eiffel, and Java. The first main contribution is a formal semantics for object-oriented programs. This semantics consists of an operational and axiomatic semantics that handles features of C#, Eiffel, and Java.

The contribution of the semantics for the subset of Java is the treatment of abrupt termination, which includes \texttt{try-catch}, \texttt{try-finally}, and \texttt{break} instructions. This subset is interesting from the semantics point of view due to the combination of \texttt{while}, \texttt{break}, \texttt{try-catch}, and \texttt{try-finally} instructions which interact in subtle ways. This formalization work highlighted some surprising aspects of Java exception semantics, of which many Java programmers and even experts are generally unaware.

The semantics for Eiffel includes exception handling, once routines, and multiple inheritance. This semantics handles the main Eiffel features, although generics are omitted. During this work, we have found that Eiffel’s exception mechanism was not ideal for formal verification. The use of \texttt{retry} instructions in a \texttt{rescue} clause complicates its verification. For this reason, a change in the Eiffel exception handling mechanism has been proposed. The change was suggested by the work described in this thesis, and will be adopted by a future revision of the language standard. As far as we know, this is the first semantics for Eiffel that includes exception handling, once routines, and multiple inheritance. We have proved soundness and completeness of the Eiffel semantics.

It is important to remark that one of the main design goals of the axiomatic semantics was to develop a Hoare-style logic that can be automatically translated to a bytecode logic. For this reason, we decided that postconditions in Hoare triples consist of a postcondition for normal termination, and a postcondition for exceptions.

Function Objects. The second main contribution is a methodology to reason about \textit{function objects} such as \texttt{delegates} in C#, \texttt{agents} in Eiffel, and \texttt{function objects} in Scala. Function objects bring a new level of abstraction to the object-oriented programming model, and require a comparable extension to specification and verification techniques. The verification methodology described in this thesis equips each function object with side-effect free routines (methods) for its pre- and postcondition, respectively. These routines can be used to specify client code relatively to the contract of the function object.

To illustrate the application of the methodology for function objects, a prototype verifier for Eiffel has been implemented. This verifier can automatically prove challenging examples proposed but left unsolved by Leavens, Leino, and Müller.

2.2.2 Proof-Transformations for Object-Oriented Programs

Proof-Transforming Compiler for Eiffel. An outcome of this work is the development of a proof-transforming compiler that translates proofs of Eiffel programs to bytecode.

\footnote{Java Hoare triples also have an extra postcondition for \texttt{break} instructions.}
2.2 Contributions

code proofs. The task of the proof-transforming compiler is made particularly challenging by the impedance mismatch between the source language, Eiffel, and the target code, .NET CIL, which does not directly support such important Eiffel mechanisms as multiple inheritance.

The Eiffel proof-transforming compiler consists of two modules: (1) a specification translator that translates Eiffel contracts to CIL contracts; and (2) a proof translator that translates Eiffel proofs to CIL proofs. The specification translator takes an Eiffel contract based on Eiffel expressions, and generates a CIL contract based on first order logic. The proof translator takes a proof in a Hoare-style logic and generates a CIL bytecode proof.

Proof-Transforming Compiler for Java. We have developed a proof-transforming compiler for a subset of Java, which handles abrupt termination. We formalize the proof translation using both CIL on .NET and JVM bytecode. The subset resulting from the combination of while, break, try-catch, and try-finally instructions raises interesting compilation issues in JVM. A try-finally instruction is compiled using code duplication: the finally block is put after the try block. If try-finally instructions are used inside of a while loop, the compilation of break instructions first duplicates the finally blocks and then inserts a jump to the end of the loop. Furthermore, the generation of exception tables in JVM is harder than the generation of .try and .catch blocks in CIL. The code duplicated before the break may have exception handlers different from those of the enclosing try block. Therefore, the exception table must be changed so that exceptions are caught by the appropriate handlers.

The formalizations of these proof-transforming compilers show that the translation of CIL is simpler than JVM bytecode due to CIL supports try-catch and try-finally instructions. The code duplication used by the JVM compiler for try-finally instructions increases the complexity of the compilation and translation functions, especially the formalization and its soundness proof.

Implementation of the Proof-Transforming Compilation Scheme. To show the practicability of the proof-transforming compilation approach, we have implemented a prototype. The implementation consists of a proof-transforming compiler for a subset of Eiffel and a proof checker. The proof checker has been formalized in Isabelle [90].
Overview and Main Results
Part II
Proofs
Program verification relies on a formal semantics of the programming language, typically a program logic such as Hoare logic \cite{Hoare}. Program logics have been developed for mainstream object-oriented languages such as Java and C#. For instance, Poetzsch-Heffter and Müller presented a Hoare-style logic for a subset of Java \cite{PoetzschHeffter}. Their logic includes the most important features of object-oriented languages such as abstract types, dynamic binding, subtyping, and inheritance. However, exception handling is not treated in their work. Although, logics that handle C# and Java’s exception handling \cite{Simpson, Groce, Sharia} have been developed, these logics do not handle `try-finally` and `break` instructions.

Eiffel has several distinctive features not present in mainstream languages, for instance, a different exception handling mechanism, once routines, and multiple inheritance. Eiffel’s once routines (methods) are used to implement global data, similar to static fields in Java. Only the first invocation of a once routine triggers an execution of the routine body; subsequent invocations return the result of the first execution. The development of formal techniques for these concepts does not only allow formally verifying Eiffel programs, but also allows comparing the different concepts, and analyzing which concepts are more suitable for formal verification.

In this chapter, we present a logic for object-oriented programs based on Poetzsch-Heffter and Müller’s logic \cite{PoetzschHeffter}. The logic is presented in three parts. First, we present the logic for the core object-oriented language, a common subset of C#, Eiffel, and Java. This logic handles object-oriented features such as object creation, attribute read and write, routine invocation, and standard instructions such as assignments, if then else, and loops.

In the second section of this chapter, the language is extended to a subset of Eiffel. This subset handles multiple inheritance, exception handling, and once routines. Agents are deferred until Chapter \ref{chapter:agents}. The third and final part of the logic is presented in Section \ref{section:java-logic}. In that section, the core language is extended to a subset of Java. The subset includes...
Java’s exception handling mechanism (try-catch, try-finally, and throw instructions) and break instructions.

The logic is sound and complete. The soundness and completeness proofs are presented in Appendix B. The chapter concludes with related work, and a discussion about the different object-oriented features that have been formalized.

This chapter is partially based on the published works [93, 80, 94]

3.1 The Core Language and its Semantics

3.1.1 The Mate Language

The source language, called Mate, is a subset of Eiffel, which includes several features also used in C# and Java. The most interesting features of the language are (1) single inheritance, (2) routine invocations, (3) attributes (fields), and (4) exception handling. Since the exception handling mechanisms for C#, Java, and Eiffel are different (C# and Java support try-catch instructions, and Eiffel supports rescue clauses), expressions in assignments can trigger exceptions but the language does not include any instruction to catch exceptions. The exception handling mechanism is extended to the Eiffel and the Java exception handling mechanisms in Section 3.2 and Section 3.3, respectively.

A Mate program is a sequence of class declarations. A class declaration consists of an optional inheritance clause, and a class body. The inheritance clause supports single inheritance. A class body is a sequence of attribute declarations or routine declarations. For simplicity, routines are functions that take always one argument, named p, and return a result. However, we do not assume that functions are side-effect free, that is, our logic fully handles heap modifications.

Figure 3.1 presents the syntax of the Mate language. Class names, routine names, variables and attributes are denoted by ClassId, RoutineId, VarId, and AttributeId, respectively. The set of variables is denoted by Var; VarId is an element of Var; list_of denotes a comma-separated list.

Boolean expressions and expressions (BoolExp and Exp) are side-effect-free, and do not trigger exceptions. In addition, we write ExpE for the expressions which might raise exceptions. For simplicity, expressions ExpE are only allowed in assignments. This assumption simplifies the presentation of the logic, especially the rules for routine invocation, if then else, and loop instructions. However, the logic could be easily extended.

The syntax of the Mate language is similar to the Eiffel syntax. In particular, the loop instruction until e loop s end iterates until the expression e evaluates to true. This instruction first evaluates the expression e, and then if this evaluation returns true,
3.1 The Core Language and its Semantics

Fig. 3.1: Syntax of the Mate Language.

The instruction $s$ is executed (opposite to the do $s$ while $e$ instruction\footnote{The do $s$ while $e$ instruction is also known as repeat $s$ until $e$ instruction (for example in Pascal).} where first $s$ is executed, and then $e$ is evaluated). The check instruction has the same semantics as the assert instruction in C# and Java. Furthermore, local variables are declared at the beginning of the routine using the keyword local.

3.1.2 The Memory Model

The state of a Mate program describes the current values of local variables, arguments, the current object, and the current object store $. A value is either a boolean, an integer, the void value, or an object reference. An object is characterized by its class and an identifier of infinite sort $ObjId$. The data type $Value$ models values; its definition is the following:
The function \( \tau : \text{Value} \rightarrow \text{Type} \) returns the dynamic type of a value. This function is defined as follows:

\[
\tau(\text{boolV } b) = \text{BoolT} \\
\tau(\text{intV } n) = \text{IntT} \\
\tau(\text{objV } cId oId) = cId \\
\tau(\text{voidV}) = \text{VoidT}
\]

The function \( \text{init} \) yields a default value for every type. The default value of boolean is \text{false}, the default value of integer is zero, and the default value of references is \text{void}. Its definition is as follows:

\[
\text{init} : \text{Type} \rightarrow \text{Value} \\
\text{init}(\text{BoolT}) = (\text{boolV } \text{false}) \\
\text{init}(\text{IntT}) = (\text{intV } 0) \\
\text{init}(\text{cId}) = \text{voidV} \\
\text{init}(\text{VoidT}) = \text{voidV}
\]

The state of an object is defined by the values of its attributes. The sort \( \text{Attribute} \) defines the attribute declaration \( T\@a \) where \( a \) is an attribute declaration in the class \( T \). We use a sort \( \text{Location} \), and a function \( \text{instvar} \) where \( \text{instvar}(V, T\@a) \) yields the instance of the attribute \( T\@a \) if \( V \) is an object reference and the object has an attribute \( T\@a \); otherwise it yields \text{undef}. The datatype definitions and the signature of \( \text{instvar} \) are the following:

\[
\text{data type} \quad \text{Attribute} = \text{Attr} \quad \text{Type} \quad \text{AttributeId} \\
\text{data type} \quad \text{Location} = \text{Loc} \quad \text{ObjId} \quad \text{AttributeId} \\
\text{instvar} : \text{Value} \times \text{Attribute} \rightarrow \text{Location} \cup \{\text{undef}\}
\]

The object store models the heap describing the states of all objects in a program at a certain point of execution. An object store is modeled by an abstract data type \( \text{ObjectStore} \). We use the object store presented by Poetzsch-Heffter and Müller \cite{107, 108}. The following operations apply to the object store: \( \text{OS}(L) \) denotes reading the location \( L \)
in the object store \( OS \); \( \text{alive}(O, OS) \) yields \( true \) if and only if the object \( O \) is allocated in the object store \( OS \); \( \text{new}(OS, C) \) yields a reference of type \( C \) to a new object in the store \( OS \); \( OS\langle L := V \rangle \) updates the object store \( OS \) at the location \( L \) with the value \( V \); \( OS(C) \) denotes the store after allocating a new object of type \( C \).

\[
\begin{align*}
\text{alive} & : \text{Value} \times \text{ObjectStore} \rightarrow \text{Bool} \\
\text{new} & : \text{ObjectStore} \times \text{ClassId} \rightarrow \text{Value} \\
\langle \_ := \_ \rangle & : \text{ObjectStore} \times \text{Location} \times \text{Value} \rightarrow \text{ObjectStore} \\
\langle \_ \rangle & : \text{ObjectStore} \times \text{ClassId} \rightarrow \text{ObjectStore}
\end{align*}
\]

In the following, we present Poetzsch-Heffter and Müller’s axiomatization \cite{107} of these functions with a brief description. The function \( \text{obj} : \text{Location} \rightarrow \text{Value} \) takes a location and yields the object it belongs to. The function \( \text{ltyp} : \text{Location} \rightarrow \text{Type} \) yields the dynamic type of a location. The function \( \text{ref} : \text{Value} \rightarrow \text{Bool} \) takes a value \( v \), and yields \( true \) if \( v \) is a reference to an object.

**Axiom 1.** Updating a location does not affect the values of other locations:
\[
\forall OS \in \text{ObjectStore}, L_1, L_2 \in \text{Location}, X \in \text{Value} : \\
L_1 \neq L_2 \Rightarrow OS\langle L_1 := X \rangle(L_2) = OS(L_2)
\]

**Axiom 2.** Reading a location updated with a value produces the same value if both the location and the value are alive:
\[
\forall OS \in \text{ObjectStore}, L \in \text{Location}, X \in \text{Value} : \\
\text{alive}(\text{obj}(L), OS) \land \text{alive}(X, OS) \Rightarrow OS\langle L := X \rangle(L) = X
\]

**Axiom 3.** Reading a location that is not alive produces the default value of the type of the location:
\[
\forall OS \in \text{ObjectStore}, L \in \text{Location} : \neg \text{alive}(\text{obj}(L), OS) \Rightarrow OS(L) = \text{init}(\text{ltyp}(L))
\]

**Axiom 4.** Updating a location that is not alive does not modify the object store:
\[
\forall OS \in \text{ObjectStore}, L \in \text{Location}, X \in \text{Value} : \neg \text{alive}(X, OS) \Rightarrow OS\langle L := X \rangle = OS
\]

**Axiom 5.** Allocating a type in the object store does not change their values:
\[
\forall OS \in \text{ObjectStore}, L \in \text{Location}, cId \in \text{ClassId} : OS\langle cId \rangle(L) = OS(L)
\]

**Axiom 6.** Updating a location does not affect the aliveness property:
\[
\forall OS \in \text{ObjectStore}, L \in \text{Location}, X, Y \in \text{Value} : \\
\text{alive}(X, OS\langle L := Y \rangle) \Leftrightarrow \text{alive}(X, OS)
\]

**Axiom 7.** An object is alive if only if the object was alive before an allocation or the object is the new object:
\[
\forall OS \in \text{ObjectStore}, X \in \text{Value}, cId \in \text{ClassId} : \\
\text{alive}(X, OS\langle cId \rangle) \Leftrightarrow \text{alive}(X, OS) \lor X = \text{new}(OS, cId)
\]
Axiom 8. Objects held by locations are alive:
\[ \forall OS \in \text{ObjectStore}, \ L \in \text{Location} : \ alive(OS(L), OS) \]

Axiom 9. Non reference values are alive:
\[ \forall OS \in \text{ObjectStore} : \ \neg \text{ref}(X) \Rightarrow alive(X, OS) \]

Axiom 10. A created object is not alive in the object store from which it was created:
\[ \forall OS \in \text{ObjectStore}, \ cId \in \text{ClassId} : \ \neg alive(new(OS, cId), OS) \]

Axiom 11. The dynamic type of a creation object of class id cId is cId:
\[ \forall OS \in \text{ObjectStore}, \ cId \in \text{ClassId} : \ \tau(new(OS, cId)) = cId \]

3.1.3 Operational Semantics

Program states are mappings from local variables and arguments to values, and from the current object store $\$ to \text{ObjectStore}. The program state is defined as follows:

\[
\begin{align*}
\text{State} & \equiv \text{Local} \times \text{Heap} \\
\text{Local} & \equiv \text{VarId} \cup \{\text{Current}, p, \text{Result}\} \rightarrow \text{Value} \cup \{\text{undef}\} \\
\text{Heap} & \equiv \{\$\} \rightarrow \text{ObjectStore}
\end{align*}
\]

\text{Local} maps local variables, the receiver object \text{Current} (\textit{this} in Java), arguments, and \text{Result} to values. Arguments are denoted by \textit{p}. The variable \text{Result} is a special variable used to store the result value, but this variable is not part of \text{VarId}. For this reason, this variable is included explicitly.

For \( \sigma \in \text{State} \), \( \sigma(e) \) denotes the evaluation of the expression \( e \) in the state \( \sigma \). Its signature is the following:
\[ \sigma : \text{ExpE} \rightarrow \text{Value} \cup \{\text{exc}\} \]

The evaluation of an expression \( e \) can yield \text{exc} meaning that an exception was triggered in \( e \). For example, \( \sigma(x/0) \) yields \text{exc}. Furthermore, the evaluation \( \sigma(y \neq 0 \land x/y = z) \) is different from \text{exc} because \( \sigma \) first evaluates \( y \neq 0 \) and then evaluates \( x/y = z \) only if \( y \neq 0 \) evaluates to \text{true}. The state \( \sigma[x := V] \) denotes the state obtained after the replacement of \( x \) by \( V \) in the state \( \sigma \).

The transitions of the operational semantics have the form:
\[ \langle \sigma, S \rangle \rightarrow \sigma', \chi \]

where \( \sigma \) and \( \sigma' \) are states, \( S \) is an instruction, and \( \chi \) is the current status of the program. The value of \( \chi \) can be either the constant \text{normal} or \text{exc}. The variable \( \chi \) is required to treat abrupt termination. The transition \( \langle \sigma, S \rangle \rightarrow \sigma', \text{normal} \) expresses that executing the instruction \( S \) in the state \( \sigma \) terminates normally in the state \( \sigma' \). The transition \( \langle \sigma, S \rangle \rightarrow \sigma', \text{exc} \) expresses that executing the instruction \( S \) in the state \( \sigma \) terminates with an exception in the state \( \sigma' \). In the following, we present the operational semantics.
3.1 The Core Language and its Semantics

Basic Instructions

Figure 3.2 presents the operational semantics for basic instructions such as assignment, compound, conditional, and loop instructions.

Assignment Instruction. The semantics for assignments consists of two rules: one when the expression $e$ throws an exception and one when it does not. In rule (3.2.1), if the expression $e$ throws an exception, then the assignment terminates with an exception and the state is unchanged. The state does not change since expressions are side-effect free. In rule (3.2.2), if $e$ does not throw any exception, after the execution of the assignment instruction, the variable $x$ is updated with the value of the expression $e$ in the state $\sigma$.

Compound. Compound is defined by two rules: in rule (3.2.3) the instruction $s_1$ is executed and an exception is triggered. The instruction $s_2$ is not executed, and the state of the compound is the state produced by $s_1$. In rule (3.2.4), $s_1$ is executed and terminates normally. The state of the compound is the state produced by $s_2$.

Check Instruction. The check instruction helps to express a property that one believes will be satisfied. If the property is satisfied then the system does not change. If the property is not satisfied then an exception is triggered. In rule (3.2.5), if the condition of the check instruction evaluates to true, then the instruction terminates normally; otherwise the check instruction triggers an exception, rule (3.2.6).

Conditional Instruction. In rule (3.2.7) the resulting state is the produced by the execution of $s_1$ since $e$ evaluates to true. In rule (3.2.8) the state produced by the execution of the conditional is the one produced by $s_2$ due to the evaluation of $e$ yields false.

Loop Instruction. In rule (3.2.9), since the until expression is true, then the body of the loop is not executed. If the until expression is false, in rule (3.2.10), the instruction $s_1$ is executed, and it triggers and exception. Thus, the state of the loop is $\sigma'$ and the status is exc. Finally, in rule (3.2.11), the condition evaluates to false, and $s_1$ terminates normally. The returned state is the one produced by the new execution of the loop.

Read Attribute Instruction. The semantics of read attribute is defined by two rules depending if the target object is void or not. In rule (3.2.12), if the value of $y$ is not Void, $x$ is updated with the value of the attribute $a$. In (3.2.13), if $y$ is Void, the instruction terminates with an exception, and the state does not change.

Write Attribute Instruction. Similar to read attribute: in rule (3.2.14) the attribute $a$ of the object $y$ is updated with the value $e$ since $y$ is not void. If $y$ is Void, the instruction terminates with an exception, and the state does not change, rule (3.2.15).

Routine Invocations

Poetzsch-Heffter and Müller [108] have developed an operational and axiomatic semantics for Java-like languages which handle inheritance, dynamic binding, subtyping, and abstract types. The source language used in their work does not handle exceptions. In this section, we extend the operational semantics for routine invocation including exception
Assignment Instruction
\[ \sigma(e) = \text{exc} \quad \frac{\langle \sigma, x := e \rangle \rightarrow \sigma, \text{exc} }{ } \quad \sigma(e) \neq \text{exc} \quad \frac{\langle \sigma, x := e \rangle \rightarrow \sigma[x := \sigma(e)], \text{normal} }{ } \]

Compound
\[ \sigma \quad \frac{\langle \sigma, s_1 \rangle \rightarrow \sigma', \text{exc} }{ } \quad \langle \sigma, s_1; s_2 \rangle \rightarrow \sigma', \text{exc} \]

Check Instruction
\[ \sigma(e) = \text{true} \quad \frac{\langle \sigma, \text{check } e \text{ end} \rangle \rightarrow \sigma, \text{normal} }{ } \quad \sigma(e) = \text{false} \quad \frac{\langle \sigma, \text{check } e \text{ end} \rangle \rightarrow \sigma, \text{exc} }{ } \]

Conditional Instruction
\[ \sigma(e) = \text{true} \quad \frac{\langle \sigma, \text{if } e \text{ then } s_1 \text{ else } s_2 \text{ end} \rangle \rightarrow \sigma', \chi }{ } \quad \sigma(e) = \text{false} \quad \frac{\langle \sigma, \text{if } e \text{ then } s_2 \text{ else } s_1 \text{ end} \rangle \rightarrow \sigma', \chi }{ } \]

Loop Instruction
\[ \sigma(e) = \text{true} \quad \frac{\langle \sigma, \text{until } e \text{ loop } s_1 \text{ end} \rangle \rightarrow \sigma, \text{normal} }{ } \quad \sigma(e) = \text{false} \quad \frac{\langle \sigma, \text{until } e \text{ loop } s_1 \text{ end} \rangle \rightarrow \sigma', \text{exc} }{ } \]

Read Attribute Instruction
\[ \sigma(y) \neq \text{voidV} \quad \frac{\langle \sigma, x := y.T@a \rangle \rightarrow \sigma[x := \sigma($)(\text{instvar}(\sigma(y), T@a))] }, \text{normal} }{ } \quad \sigma(y) = \text{voidV} \quad \frac{\langle \sigma, x := y.T@a \rangle \rightarrow \sigma, \text{exc} }{ } \]

Write Attribute Instruction
\[ \sigma(y) \neq \text{voidV} \quad \frac{\langle \sigma, y.T@a := e \rangle \rightarrow \sigma[$ := \sigma($)(\text{instvar}(\sigma(y), T@a) := \sigma(e))] }, \text{normal} }{ } \quad \sigma(y) = \text{voidV} \quad \frac{\langle \sigma, y.T@a := e \rangle \rightarrow \sigma, \text{exc} }{ } \]

Fig. 3.2: Operational Semantics for Basic Instructions
3.1 The Core Language and its Semantics

Poetzsch-Heffter and Müller distinguish between virtual routines and routine implementation. A class $T$ has a virtual routine $T:m$ for every routine $m$ that it declares or inherits. A class $T$ has a routine implementation $T@m$ for every routine $m$ that it defines (or redefines). We assume in the following that every invocation is decorated with the virtual routine being invoked. The semantics of routine invocations uses two functions: $body$ and $impl$. The function $impl(T, m)$ yields the implementation of routine $m$ in class $T$. This implementation could be defined by $T$ or inherited from a superclass. The function $body$ yields the instruction constituting the body of a routine implementation. The signatures of these functions are as follows:

$$impl : Type \times RoutineId \to RoutineImpl \cup \{undef\}$$

$$body : RoutineImpl \to Instr$$

The operational semantics of routine invocation is defined with the following rules:

$$\sigma(y) = voidV \quad \langle \sigma, x := y, T:m(e) \rangle \to \sigma, \text{exc} \quad (3.1)$$

$$\sigma(y) \neq voidV \quad \langle \text{initSt}[Current := \sigma(y), p := \sigma(e), $ := \sigma($)], body(impl(\tau(\sigma(y)), m))] \to \sigma', \text{normal} \quad (3.2)$$

$$\sigma(y) \neq voidV \quad \langle \text{initSt}[Current := \sigma(y), p := \sigma(e), $ := \sigma($)], body(impl(\tau(\sigma(y)), m))] \to \sigma', \text{exc} \quad (3.3)$$

In rule (3.1), since the target $y$ is Void, the state $\sigma$ is not changed and an exception is triggered. In rules (3.2) and (3.3), the target is not Void, thus, the Current object is updated with $y$, and the argument $p$ by the expression $e$, and then the body of the routine is executed. To handle dynamic dispatch, first, the dynamic type of $y$ is obtained using the function $\tau$. Then, the routine implementation is determined by the function $impl$. Finally, the body of the routine is obtained by the function $body$. If the routine $m$ terminates normally, then $x$ is updated with the result of $m$ (rule (3.2)). If an exception is triggered in $m$, $x$ is not updated, and the state is $\sigma'$ (rule (3.3)).

Local Variables Declaration

Local variables, as well as the Result variable, have default initialization values. Thus, the operational semantics first initialize the local variables and Result and then it executes the body of the routine. Given a routine declaration $rId$ (contracts omitted):
the function \texttt{body} returns the following result:

\begin{verbatim}
local  v_1 : T_1; ... v_i : T_i; Result : T'; s
\end{verbatim}

Note that the function \texttt{body} adds the declaration of the variable \texttt{Result}. This declaration allows proving properties about the \texttt{Result}. In particular, it allows initializing the variable \texttt{Result} with its initial value.

Local variables are initialized using rule (3.4). The values of the variables \( v_1 \ldots v_n \) are updated with their default value according to their types. The function \texttt{init}, given a type \( T \) returns its default value; \texttt{init(INTEGER)} returns 0; \texttt{init(BOOLEAN)} returns \texttt{false} and \texttt{init(T)} where \( T \) is a reference type returns \texttt{Void}. The rule is the following:

\begin{equation}
\langle \sigma[v_1 := \texttt{init}(T_1), \ldots, v_n := \texttt{init}(T_n)], s \rangle \rightarrow \sigma', \chi
\end{equation}

(3.4)

Creation Instruction

The creation instruction is similar to the Eiffel creation instruction. First, a new object of type \( T \) is created and assigned to \texttt{Current}. The current object store \( $ \) is updated with the store \( \sigma($)(T) \) and the argument \( p \) is updated with the expression \( e \). Then, the routine \texttt{make} is invoked. Similar to the invocation rule, the body of the routine \texttt{make} is obtained using the function \texttt{body}. The semantics is defined with the following rules:

\begin{equation}
\langle \sigma, x := \texttt{create T.make(e)} \rangle \rightarrow \sigma[x := \sigma'(\texttt{Current}), $ := \sigma($)], normal
\end{equation}

(3.5)

\begin{equation}
\langle \sigma, x := \texttt{create T.make(e)} \rangle \rightarrow \sigma[$ := \sigma($)], exc
\end{equation}

(3.6)

In rule (3.5), the routine \texttt{make} terminates normally, and the object \( x \) is updated with the \texttt{Current} object in \( \sigma' \). In Mate, when a new object is created, its attributes are initialized with the default value. In the semantics, this is done by the function \texttt{new} which creates the new object initializing its attributes. If the routine \texttt{make} triggers an exception (rule 3.6), the state produced by the creation instruction is \( \sigma' \), and \( x \) is not updated.
3.1 The Core Language and its Semantics

3.1.4 A Programming Logic for Mate

The logic for Mate is based on the programming logic presented by Poetzsch-Heffter and Müller [108, 109]. We take over many of the rules, especially all the language-independent rules such as the rules of consequence. Poetzsch-Heffter et al. [109] uses a special variable \( \chi \) to capture the status of the program such as normal or exceptional status. We instead use Hoare triples with two postconditions to encode the status of the program execution.

The logic is a Hoare-style logic. Properties of routines and routine bodies are expressed by Hoare triples of the form \( \{ P \} \ S \{ Q_n , Q_e \} \), where \( P, Q_n, Q_e \) are formulas in first order logic, and \( S \) is a routine or an instruction. The third component of the triple consists of a normal postcondition (\( Q_n \)), and an exceptional postcondition (\( Q_e \)).

The triple \( \{ P \} S \{ Q_n , Q_e \} \) defines the following refined partial correctness property: if \( S \)'s execution starts in a state satisfying \( P \), then (1) \( S \) terminates normally in a state where \( Q_n \) holds, or (2) \( S \) throws an exception and \( Q_e \) holds, or (3) \( S \) aborts due to errors or actions that are beyond the semantics of the programming language, e.g., memory allocation problems, or (4) \( S \) runs forever.

Since one of our design goals of the logic is to be able to automatically translate it to a bytecode logic, and the bytecode logic supports partial correctness, we restrict the source logic to partial correctness. Extending the source logic, the bytecode logic, and the proof transformation to full functional correctness is part of future work.

Boolean Expressions

Preconditions and postconditions are formulas in first order logic (FOL). We encode the programming language expressions in FOL using a shallow embedding. This encoding is the classical encoding of FOL; thus it is omitted. Since expressions in the programming language might trigger exceptions, we introduce a function \( \text{safe} \) that takes the expression, and yields a safe expression. A safe expression is an expression whose evaluation does not trigger an exception. Thus, the encoding takes a safe expression in the programming language and produces a FOL formula such that:

\[
\text{embed}(e)(\sigma) \iff \sigma(e) = \text{boolV true}
\]

This embedding is trivial, thus it is omitted.

**Definition 1** (Safe Expression). An expression \( e \) is a safe expression if and only if \( \forall \sigma : \sigma(e) \neq \text{exc} \).

**Definition 2** (Function Safe). The function \( \text{safe} : \text{ExpE} \to \text{Exp} \) returns an expression that expresses if the input expression is safe or not. The definition of this function is the
following:

\[
\begin{align*}
\text{safe} &: \text{ExpE} \rightarrow \text{Exp} \\
\text{safe} (e_1 \text{ oper } e_2) &= \text{safe}(e_1) \text{ and } \text{safe}(e_2) \text{ and } \text{safe}_{\text{op}} (\text{oper}, e_1, e_2) \\
\text{safe} (e) &= \text{true} \\
\text{safe}_{\text{op}} &: \text{Op} \times \text{ExpE} \times \text{ExpE} \rightarrow \text{Exp} \\
\text{safe}_{\text{op}} (\text{/}/, e_1, e_2) &= (e_2 / = 0) \\
\text{safe}_{\text{op}} (\text{oper}, e_1, e_2) &= \text{true}
\end{align*}
\]

Lemma 1. For each expression \( e \), \( \text{safe}(e) \) satisfies:

\[ \sigma(\text{safe}(e)) = \text{true} \iff \sigma(e) \neq \text{exc} \]

Proof. By induction on the structure of \( e \), and definition of \( \text{safe} \).

\[ \square \]

Lemma 2 (Substitution). If the expression \( e \) is a \textit{safe} expression, then:

\[ \forall \sigma : (\sigma \models P[e/x] \iff \sigma[x := \sigma(e)] \models P) \]

Proof. By induction on the structure of \( P \), and definition of \( \models \).

\[ \square \]

We define \( \sigma \models P \) as the usual definition of \( \models \) in first order logic.

Signatures of Contracts

Contracts refer to attributes, variables, and types. We introduce a signature \( \Sigma \) that represents the constant symbols of these entities. Given a Mate program, \( \Sigma \) denotes the signature of sorts, functions, and constant symbols as described in Section 3.1.1. Arguments, program variables, and the current object store \( \$ \) are treated syntactically as constants of \texttt{Value} and \texttt{ObjectStore}. Preconditions and postconditions of Hoare triples are formulas over \( \Sigma \cup \{\text{Current}, p, \text{Result}\} \cup \text{Var}(r) \) where \( r \) is a routine, and \( \text{Var}(r) \) denotes the set of local variables of \( r \). Note that we assume \( \text{Var}(r) \) does not include the \texttt{Result} variable, it only includes the local variables declared by the programmer. Routine preconditions are formulas over \( \Sigma \cup \{\text{Current}, p, \$\} \), routine postconditions for normal termination are formulas over \( \Sigma \cup \{\text{Result}, \$\} \), and routine exceptional postconditions are formulas over \( \Sigma \cup \$ \).

We treat recursive routines in the same way as Poetzsch-Heffter and Müller [108]. We use \textit{sequents} of the form \( \mathcal{A} \vdash \mathcal{A} \) where \( \mathcal{A} \) is a set of routine annotations and \( \mathcal{A} \) is a triple. Triples in \( \mathcal{A} \) are called assumptions of the sequent, and \( \mathcal{A} \) is called the consequent of the sequent. Thus, a sequent expresses that we can prove a triple based on the assumptions about routines. In the following, we present the logic for Mate.
3.1 The Core Language and its Semantics

Basic Rules

Figure 3.3 presents the axiomatic semantics for basic instructions such as assignment, compound, loop, and conditional instructions. In the assignment rule, if the expression $e$ is safe (it does not throw any exception) then the precondition is obtained by replacing $x$ by $e$ in $P$. Otherwise the precondition is the exceptional postcondition. In the compound rule, first the instruction $s_1$ is executed. If $s_1$ triggers an exception, $s_2$ is not executed, and $R_e$ holds. If $s_1$ terminates normally, $s_2$ is executed, and the postcondition of the compound is the postcondition of $s_2$.

In the conditional rule, the instruction $s_1$ is executed if $e$ evaluates to $true$, and the result of the conditional is the postcondition of $s_1$. If $e$ evaluates to $false$, $s_2$ is executed, and the result of the conditional is the postcondition of $s_2$. In the check axiom, if the condition evaluates to $true$, then the instruction terminates normally and $P \land e$ holds. If $e$ is $false$, an exception is triggered and $P \land \neg e$ holds. Note that for conditionals, check instructions, and loops, the expression $e$ is syntactically restricted not to trigger an exception, which simplifies the rules significantly.

In the loop rule, if the until expression $e$ does not hold, the body of the loop ($s_1$) is executed. If $s_1$ finishes in normal execution then the invariant $I$ holds, and if $s_1$ throws an exception, $R_e$ holds. In the read attribute axiom, if $y$ is not $Void$, the value of the attribute $a$ defined in the class $T$ of the object $y$ is assigned to $x$. Otherwise, an exception is triggered and $Q_e$ holds. Similar to read attribute, in the write attribute axiom, if $y$ is not $Void$, the attribute $a$ defined in the class $T$ of the object $y$ is updated with the expression $e$. Otherwise, an exception is triggered and $Q_e$ holds.

Routine Invocation

Routine invocations are verified based on properties of the virtual routine being called. A routine specification for $T:m$ reflects the common properties of all implementations that might be executed on invocation of $T:m$. In the routine invocation rule, if the target $y$ is not $Void$, the current object is replaced by $y$ and the formal parameter $p$ by the expression $e$ in the precondition $P$. Then, in the postcondition $Q_n$, $Result$ is replaced by $x$ to assign the result of the invocation. If $y$ is $Void$ the invocation triggers and exception, and $Q_e$ holds.

Routine Invocation Rule:

$$ \mathcal{A} \vdash \{ \ P \ \} \ T:m \ \{ \ Q_n \ , \ Q_e \ \} $$

$$ \mathcal{A} \vdash \left\{ \begin{array}{l}
(y \neq Void \land P[y/Current,e/p]) \lor \\
(y = Void \land Q_e)
\end{array} \right\} \ x := y.T:m(e) \ \{ \ Q_n[x/Result] \ , \ Q_e \ \} $$

The following rule expresses the fact that local variables different from the left-hand side variable are not modified by an invocation. This rule allows one to substitute logical variables $Z$ in preconditions and postconditions by local variables $w$ ($w$ different from $x$).
Assignment Axiom:
\[ A \vdash \{ (\text{safe}(e) \land P[e/x]) \lor (\neg\text{safe}(e) \land Q_e) \} \quad x := e \{ P, Q_e \} \]

Compound Rule:
\[ A \vdash \{ P \} \quad s_1 \{ Q_n, R_e \} \quad A \vdash \{ Q_n \} \quad s_2 \{ R_n, R_e \} \]
\[ A \vdash \{ P \} \quad s_1 ; s_2 \{ R_n, R_e \} \]

Conditional Rule:
\[ A \vdash \{ P \land e \} \quad s_1 \{ Q_n, Q_e \} \quad A \vdash \{ P \land \neg e \} \quad s_2 \{ Q_n, Q_e \} \]
\[ A \vdash \{ P \} \quad \text{if } e \text{ then } s_1 \text{ else } s_2 \text{ end} \{ Q_n, Q_e \} \]

Check Axiom:
\[ A \vdash \{ P \} \quad \text{check } e \text{ end} \{ (P \land e), (P \land \neg e) \} \]

Loop Rule:
\[ A \vdash \{ \neg e \land I \} \quad s_1 \{ I, R_e \} \]
\[ A \vdash \{ I \} \quad \text{until } e \text{ loop } s_1 \text{ end} \{ (I \land e), R_e \} \]

Read Attribute Axiom:
\[ A \vdash \{ (y \neq \text{Void} \land P[\langle \text{instvar}(y, T@a) \rangle/x]) \lor (y = \text{Void} \land Q_e) \} \quad x := y.T@a \{ P, Q_e \} \]

Write Attribute Axiom:
\[ A \vdash \{ (y \neq \text{Void} \land P[\langle \text{instvar}(y, T@a) := e \rangle/x]) \lor (y = \text{Void} \land Q_e) \} \quad y.T@a := e \{ P, Q_e \} \]

Fig. 3.3: Axiomatic Semantics for Basic Instructions

Invoc-var-rule:
\[ A \vdash \{ P \} \quad x := y.T:m(e) \{ Q_n, Q_e \} \]
\[ A \vdash \{ P[w/Z] \} \quad x := y.T:m(e) \{ Q_n[w/Z], Q_e[w/Z] \} \]

To prove a triple for a virtual routine \( T:m \), one has to derive the property for all possible implementations, that is, \( \text{impl}(m, T) \) and \( S:m \) for all subclasses \( S \) of \( T \). The corresponding rule is the following:
3.1 The Core Language and its Semantics

**Class Rule:**

\[
A \vdash \{ \tau(\text{Current}) = T \land P \} \impl(T, m) \{ Q_n, Q_e \}
\]

\[
A \vdash \{ \tau(\text{Current}) \prec T \land P \} \quad T:m \{ Q_n, Q_e \}
\]

\[
A \vdash \{ \tau(\text{Current}) \preceq T \land P \} \quad T:m \{ Q_n, Q_e \}
\]

In the subtype rule, if \( S \) is a subtype of \( T \), an invocation of \( T:m \) on an object of type \( S \) is equivalent to an invocation of \( S:m \). The function \( \preceq \) denotes the subtype relation, and \( \prec \) denotes the irreflexive subtype relation where \( S \prec T \iff_{\text{def}} S \preceq T \land S \neq T \).

**Subtype Rule:**

\[
A \vdash \{ P \} \quad S \preceq T \quad S:m \{ Q_n, Q_e \}
\]

\[
A \vdash \{ \tau(\text{Current}) \preceq S \land P \} \quad T:m \{ Q_n, Q_e \}
\]

**Routine Implementation**

The following rule is used to derive properties of routine implementations from their bodies. To handle recursion, the assumption \( \{ P \} \quad T@m \{ Q_n, Q_e \} \) is added to the set of routine annotations \( A \).

**Routine Implementation Rule:**

\[
A, \{ P \} \quad T@m \{ Q_n, Q_e \} \vdash \{ P \} \quad \text{body}(T@m) \{ Q_n, Q_e \}
\]

\[
A \vdash \{ P \} \quad T@m \{ Q_n, Q_e \}
\]

**Local**

In Mate, local variables have default values. To initialize local variables we use the function \( \text{init} \). The following rule is used to initializes default values:

**Local Rule:**

\[
A \vdash \{ P \land v_1 = \text{init}(T_1) \land \ldots \land v_n = \text{init}(T_n) \} \quad s \{ Q_n, Q_e \}
\]

\[
A \vdash \{ P \land \text{local} \ v_1 : T_1; \ldots \ v_n : T_n; \ s \} \{ Q_n, Q_e \}
\]

where \( P, Q_n, \) and \( Q_e \) are routine specifications.
Creation

The creation instruction creates an object of type $T$ and then invokes the routine $\text{make}$. In the following rule, the object creation is expressed by the replacement $\text{new}(\$, T)/\text{Current}, \$\langle T \rangle/\$. The replacement $e/p$ is added due to the routine invocation. Finally, in the postcondition, the $\text{Current}$ object is replaced by $x$. The rule is the following:

$$
\text{Creation Rule:}
$$

$$
\mathcal{A} \vdash \{ P \} \quad \text{make} \quad \{ Q_n, Q_e \}
$$

$$
\mathcal{A} \vdash \{ P \left[ \frac{\text{new}(\$, T)/\text{Current}, \$\langle T \rangle/\$}{e/p} \right] \} \quad x := \text{create} \{ T \}.\text{make}(e) \quad \{ Q_n[x/\text{Current}], Q_e \}
$$

Language-Independent Rules

The rules we have presented in the above sections depend from the specific instructions and features of the programming language. Figure 3.4 presents rules that can be applied to any programming language. The $\text{false axiom}$ allows us to prove anything assuming $\text{false}$. The $\text{assumpt-axiom}$ allows proving a triple $A$ assuming the triple holds. The $\text{assumpt.intro-axiom}$ and the $\text{assumpt.elim-axiom}$ allow introducing and eliminating a triple $A_0$ in the hypothesis.

The $\text{strength rule}$ allows proving a Hoare triple with a stronger precondition if the precondition $P'$ implies the precondition $P$, and the Hoare triple can be proved using the precondition $P$. The $\text{weak rule}$ is similar but it weakens the postcondition. This rule can be used to weaken both the normal postcondition $Q_n$, and the exceptional postcondition $Q_e$. The $\text{conjunction}$ and $\text{disjunction rule}$, given the two proofs for the same instruction but using possible different pre- and postconditions, it concludes the conjunction and disjunction of the pre- and postcondition respectively. The $\text{invariant rule}$ conjuncts $W$ in the precondition and postcondition assuming that $W$ does not contain neither program variables or $\$$. The $\text{substitution rule}$ substitutes $Z$ by $t$ in the precondition and postcondition. Finally, the $\text{all-rule}$ and $\text{ex-rule}$ introduces universal and existential quantifiers respectively.

3.1.5 Example

In this section, we illustrate the application of the programming logic for the Mate language. The function $\text{sum}$ is defined in the class $\text{MATH}$, and it implements a function that returns the sum from 1 to $n$ where $n > 1$. Its postcondition is $\text{Result} = (n \ast (n + 1))/2$. To simplify the proof, we assume that the class $\text{MATH}$ does not have descendants. The implementation is as follows:
### 3.1 The Core Language and its Semantics

#### Assumpt-axiom

\[ \mathcal{A} \vdash A \]

#### Assumpt-intro-axiom

\[ \mathcal{A}, A \vdash A \]

#### Assumpt-elim-axiom

\[ \mathcal{A} \vdash A \]

#### Strength

\[ P' \Rightarrow P \]

\[ \mathcal{A} \vdash \{ P \} \quad s_1 \quad \{ Q_n, Q_e \} \]

\[ \mathcal{A} \vdash \{ P' \} \quad s_1 \quad \{ Q_n, Q_e \} \]

\[ \mathcal{A} \vdash \{ P' \land P'' \} \quad s_1 \quad \{ Q_n \land Q'_n, Q_e \land Q'_e \} \]

#### False axiom

\[ \vdash \{ \text{false} \} \quad s_1 \quad \{ \text{false}, \text{false} \} \]

#### Assumpt-intro-axiom

\[ \mathcal{A} \vdash A_0 \]

\[ \mathcal{A}, A_0 \vdash A \]

#### Assumpt-elim-axiom

\[ \mathcal{A} \vdash A \]

#### Weak

\[ \mathcal{A} \vdash \{ P \} \quad s_1 \quad \{ Q_n, Q_e \} \]

\[ Q_n \Rightarrow Q'_n \]

\[ Q_e \Rightarrow Q'_e \]

\[ \mathcal{A} \vdash \{ P \} \quad s_1 \quad \{ Q_n, Q_e \} \]

#### Disjunction

\[ \mathcal{A} \vdash \{ P' \} \quad s_1 \quad \{ Q'_n, Q'_e \} \]

\[ \mathcal{A} \vdash \{ P'' \} \quad s_1 \quad \{ Q''_n, Q''_e \} \]

\[ \mathcal{A} \vdash \{ P' \lor P'' \} \quad s_1 \quad \{ Q'_n \lor Q''_n, Q'_e \lor Q''_e \} \]

#### Conjunction

\[ \mathcal{A} \vdash \{ P' \} \quad s_1 \quad \{ Q'_n, Q'_e \} \]

\[ \mathcal{A} \vdash \{ P'' \} \quad s_1 \quad \{ Q''_n, Q''_e \} \]

\[ \mathcal{A} \vdash \{ P' \land P'' \} \quad s_1 \quad \{ Q'_n \land Q''_n, Q'_e \land Q''_e \} \]

#### Substitution

\[ \mathcal{A} \vdash \{ P \} \quad s_1 \quad \{ Q_n, Q_e \} \]

\[ \mathcal{A} \vdash \{ P[t/Z] \} \quad s_1 \quad \{ Q_n[t/Z], Q_e[t/Z] \} \]

where \( Z \) is an arbitrary logical variable and \( t \) a \( \Sigma \)-term.

#### Invariant

\[ \mathcal{A} \vdash \{ P \} \quad s_1 \quad \{ Q_n, Q_e \} \]

\[ \mathcal{A} \vdash \{ P \land W \} \quad s_1 \quad \{ Q_n \land W, Q_e \land W \} \]

where \( W \) is a \( \Sigma \)-formula, i.e. does not contain program variables or $s$.

#### all-rule

\[ \mathcal{A} \vdash \{ P[Y/Z] \} \quad s_1 \quad \{ Q_n, Q_e \} \]

\[ \mathcal{A} \vdash \{ P[Y/Z] \} \quad s_1 \quad \{ \forall Z : Q_n, \forall Z : Q_e \} \]

where \( Z, Y \) are arbitrary, but distinct logical variables.

#### ex-rule

\[ \mathcal{A} \vdash \{ P \} \quad s_1 \quad \{ Q_n[Y/Z], Q_e[Y/Z] \} \]

\[ \mathcal{A} \vdash \{ \exists Z : P \} \quad s_1 \quad \{ Q_n[Y/Z], Q_e[Y/Z] \} \]

where \( Z, Y \) are arbitrary, but distinct logical variables.

---

**Fig. 3.4: Language-Independent Rules**
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 \geq i and i > 1
until
  i = n+1
loop
  Result := Result + i
  i := i+1
end
ensure
  Result = (n*(n+1))/2
end

Applying the logic, we have proven that the routine \textit{sum} satisfies the following specifications:

\[
\{ n > 1 \} \quad \text{MATH:sum} \quad \{ Result = (n \ast (n + 1))/2 , \ false \}
\]

To be able to prove the specifications for the virtual routine \textit{MATH:sum}, we first prove the same specification for the routine implementation \textit{MATH@sum}. Figure 3.5 presents a sketch of the proof for:

\[
\{ n > 1 \} \quad \text{body(MATH@sum)} \quad \{ Result = (n \ast (n + 1))/2 , \ false \}
\]

In the proof of Figure 3.5, we have applied the assignment, loop, and compound rule, and the weak and strength rules. The most interesting part of the proof is the application of the weak and the strength rule. In lines 13-15, one shows that the invariant of the loop holds at the entry of the loop using the weak rule. In lines 21-23, we show that the loop invariant implies the weakest precondition of the loop body, applying the strength rule.

Applying the routine implementation rule (presented on page 31), one can prove that the routine implementation \textit{MATH@sum} satisfies its specification:

\[
\{ \tau(\text{Current}) = \text{MATH} \land n > 1 \} \quad \text{body(MATH@sum)} \quad \{ Result = (n \ast (n + 1))/2 , \ false \}
\]

\[
\{ \tau(\text{Current}) = \text{MATH} \land n > 1 \} \quad \text{MATH@sum} \quad \{ Result = (n \ast (n + 1))/2 , \ false \}
\]

The class \textit{MATH} does not have any descendent. Thus, we can show:
3.1 The Core Language and its Semantics

\textit{sum} \,(n: \text{INTEGER}): \text{INTEGER}

\textbf{require}
\text{positive : } n > 1

\textbf{local}
\hspace{1em} i: \text{INTEGER}

\textbf{do}
\hspace{1em} \{ n > 1 \}

\textbf{from}
\hspace{1em} \{ n > 1 \}
\hspace{1em} \text{Result} := 1
\hspace{1em} \{ n > 1 \ \text{and} \ \text{Result} = 1 \}
\hspace{1em} i := 2
\hspace{1em} \{ n > 1 \ \text{and} \ \text{Result} = 1 \ \text{and} \ i = 2 \}
\hspace{1em} \Rightarrow \ [\text{Weak Rule}]
\hspace{1em} \{ \text{Result} = ((i-1)*i)/2 \ \text{and} \ n+1 \geq i \ \text{and} \ i > 1 \}

\textbf{invariant}
\hspace{1em} \text{Result} = ((i-1)*i)/2 \ \text{and} \ n+1 \geq i \ \text{and} \ i > 1

\textbf{until}
\hspace{1em} i = n+1

\textbf{loop}
\hspace{1em} \{ i \neq n+1 \ \text{and} \ \text{Result} = ((i-1)*i)/2 \ \text{and} \ n+1 \geq i \ \text{and} \ i > 1 \}
\hspace{1em} \Rightarrow \ [\text{Strength Rule}]
\hspace{1em} \{ i \neq n+1 \ \text{and} \ \text{Result} + i = (i*(i+1))/2 \ \text{and} \ n+1 \geq i \ \text{and} \ i > 1 \}
\hspace{1em} \text{Result} := \text{Result} + i
\hspace{1em} \{ i \neq n+1 \ \text{and} \ \text{Result} = (i*(i+1))/2 \ \text{and} \ n+1 \geq i \ \text{and} \ i > 1 \}
\hspace{1em} i := i+1
\hspace{1em} \{ \text{Result} = ((i-1)*i)/2 \ \text{and} \ n+1 \geq i \ \text{and} \ i > 1 \}

\textbf{end}

\textbf{ensure}
\hspace{1em} \text{Result} = (n*(n+1))/2

\textbf{end}

Fig. 3.5: Proof of the \textit{sum} Function
\[ \tau(\text{Current}) \prec \text{MATH} \land n > 1 \Rightarrow \text{false} \]

Then, applying the strength rule we show:

\[
\{ \text{false} \} \quad \text{MATH:sum} \quad \{ \text{Result} = (n \times (n + 1))/2 \land \text{false} \}
\]

\[
\{ \tau(\text{Current}) \prec \text{MATH} \land n > 1 \} \quad \text{MATH:sum} \quad \{ \text{Result} = (n \times (n + 1))/2 \land \text{false} \}
\]

Since \text{sum} is implemented in the class \text{MATH}, we have \text{impl(MATH, sum)}=\text{MATH@sum}.

Therefore, we can conclude the proof by applying the class rule (presented on page 29) using the above two proofs:

\[
\{ \tau(\text{Current}) \prec \text{MATH} \land n > 1 \} \quad \text{MATH:sum} \quad \{ \text{Result} = (n \times (n + 1))/2 \land \text{false} \}
\]

\[
\{ \tau(\text{Current}) = \text{MATH} \land n > 1 \} \quad \text{impl(MATH, sum)} \quad \{ \text{Result} = (n \times (n + 1))/2 \land \text{false} \}
\]

\[
\{ \tau(\text{Current}) \preceq \text{MATH} \land n > 1 \} \quad \text{MATH:sum} \quad \{ \text{Result} = (n \times (n + 1))/2 \land \text{false} \}
\]

3.2 A Logic for Eiffel

In Section 3.1, we have presented an operational and axiomatic semantics for an object-oriented language which handles basic instructions, single inheritance, and dynamic binding. In the Mate language, we assume assignments might trigger exceptions, but there is no instruction to catch them. In this section, we extend the language to a subset of Eiffel [72]. In particular, the subset of Eiffel supports exception handling using the \text{rescue} clause.

3.2.1 The Eiffel Language

The subset of Eiffel includes the most important features, although agents are omitted (a methodology to reason about agents is presented in Chapter 4). The most interesting supported concepts are: (1) multiple inheritance, (2) exception handling, and (3) once routines. Figure 3.6 presents the syntax of the subset of Eiffel.

In Eiffel, contracts are checked at runtime. One of the design goals of our logic is that programs behave in the same way when contracts are checked at runtime and when they are not. For this reason, we demand that expressions occurring in contracts to be side-effect-free, and to not trigger exceptions.

3.2.2 Operational Semantics

The memory model used in the semantics of the subset of Eiffel is the same memory model presented in Section 3.1.2. The program state is extended as follows:
3.2 A Logic for Eiffel

Program ::= ClassDecl∗
ClassDecl ::= class ClassId [Inheritance] ClassBody end
Type ::= BoolT | IntT | ClassId | VoidT
Inheritance ::= inherit Parent+
Parent ::= Type [Undefine] [Redefine] [Rename]
Undefine ::= undefine list_of RoutineId
Redefine ::= redefine list_of RoutineId
Rename ::= rename list_of (RoutineId as RoutineId)
ClassBody ::= MemDecl∗
MemDecl ::= AttributeId Type | Routine
Routine ::= RoutineId (Type): Type

[local list_of (VarId : Type)]
[do | once]
Instr
[rescue Instr]
end

Instr, Exp,
ExpE, BoolExp ::= //as defined in Figure 3.1

Fig. 3.6: Syntax of the Subset of Eiffel.

State ≡ Local × Heap
Local ≡ VarId ∪ {Current, p, Result, Retry} → Value ∪ {undef}
Heap ≡ {§} → ObjectStore

The Retry variable is a special variable used to store the retry value but this variable is not part of VarId. For this reason, this variable is included explicitly. The transitions of the operational semantics have the same for as defined in Section 3.1.3:

⟨σ, S⟩ → σ′, χ

Exception Handling

Exceptions [73] raise some of the most interesting problems addressed in this chapter. A routine execution either succeeds - meaning that it terminates normally - or fails, triggering an exception. An exception is an abnormal event, which occurred during the execution. To treat exceptions, each routine may contain a rescue clause. If the routine body is executed and terminates normally, the rescue clause is ignored. However, if the routine body triggers an exception, control is transferred to the rescue clause. Each routine implicitly defines a boolean local variable Retry (in a similar way as for Result). If at the end of the clause the variable Retry has value true, the routine body (do clause) is executed again. Otherwise, the routine fails, and propagates the exception. If the rescue clause triggers another exception, the exception is propagated, and it can be handled
through the `rescue` clause of the caller. The `Retry` variable can be assigned to in either a `do` clause or a `rescue` clause.

This specification slightly departs from the current Eiffel standard, where `retry` is an instruction, not a variable. The change was suggested in our previous work \[93, 96\] and will be adopted by a future revision of the language standard.

The operational semantics for the exception handling mechanism is defined by rules (3.7)-(3.10) below. If the execution of `s1` terminates normally, then the `rescue` clause is not executed, and the returned state is the one produced by `s1` (rule 3.7). If `s1` terminates with an exception, and `s2` triggers another exception, the `rescue` terminates in an exception returning the state produced by `s2` (rule 3.8). If `s1` triggers an exception and `s2` terminates normally, but the `Retry` variable is `false`, then the `rescue` terminates with an exception returning the state produced by `s2` (rule 3.9). In the analogous situation with `Retry` being `true`, the `rescue` is executed again and the result is the one produced by the new execution of the `rescue` (rule 3.10). Note that the `rescue` is a loop that iterates over `s2`; `s1` until `s1` terminates normally or `Retry` is `false`.

\[
\begin{align*}
\langle \sigma, s_1 \rangle \rightarrow \sigma', \text{normal} \\
\langle \sigma, s_1 \text{ rescue } s_2 \rangle \rightarrow \sigma', \text{normal} \\
\langle \sigma, s_1 \rangle \rightarrow \sigma', \text{exc} \\
\langle \sigma', s_2 \rangle \rightarrow \sigma'', \text{exc}
\end{align*}
\]

(3.7)

\[
\begin{align*}
\langle \sigma, s_1 \rangle \rightarrow \sigma', \text{exc} \\
\langle \sigma', s_2 \rangle \rightarrow \sigma'', \text{exc}
\end{align*}
\]

(3.8)

\[
\begin{align*}
\langle \sigma, s_1 \rangle \rightarrow \sigma', \text{exc} \\
\langle \sigma', s_2 \rangle \rightarrow \sigma'', \text{normal} \\
\neg \sigma''(\text{Retry})
\end{align*}
\]

(3.9)

\[
\begin{align*}
\langle \sigma, s_1 \rangle \rightarrow \sigma', \text{exc} \\
\langle \sigma', s_2 \rangle \rightarrow \sigma'', \text{normal} \\
\sigma''(\text{Retry})
\end{align*}
\]

(3.10)

Once Routines

The mechanism used in Eiffel to access a shared object is `once routines`. This section focuses on `once` functions; `once` procedures are similar. The semantics of `once` functions is as follows. When a `once` function is invoked for the first time in a program execution, its body is executed and the outcome is cached. This outcome may be a result in case the body terminates normally or an exception in case the body triggers an exception. For subsequent invocations, the body is not executed; the invocations produce the same outcome (result or exception) like the first invocation.

Note that whether an invocation is the first or a subsequent one is determined solely by the routine implementation name. Thus, if a `once` routine `m` is implemented in class `C`, descendants of `C` that do not redefine `m` access to the same shared objects. For example,
let the dynamic type of \( x \) be \( C \), and the dynamic type of \( y \) be a subtype of \( C \). If the first invocation \( x.m \) returns 1, subsequent invocations \( y.m \) will also return 1.

To be able to develop a semantics for once functions, we also need to consider recursive invocations. As described in the Eiffel ECMA standard [75], a recursive call may start before the first invocation finished. In that case, the recursive call will return the result that has been obtained so far. The mechanism is not so simple. For example the behavior of the following recursive factorial function might be surprising:

\[
\text{factorial} \ (i: \text{INTEGER}) : \text{INTEGER} \\
\text{--- A once factorial function.} \\
\text{require} \\
\quad i \geq 0 \\
\text{once} \\
\quad \text{if } i \leq 1 \text{ then} \\
\quad \quad \text{Result} := 1 \\
\quad \text{else} \\
\quad \quad \text{Result} := i \\
\quad \quad \text{Result} := \text{Result} \ast \text{factorial} \ (i - 1) \\
\quad \text{end} \\
\text{end}
\]

This example is a typical factorial function but it is also a once function, and the assignment \( \text{Result} := i \ast \text{factorial}(i - 1) \) is split into two separate assignments. If one invokes \( \text{factorial}(3) \) we observe that the returned result is 9. The reason is that the first invocation, \( \text{factorial}(3) \), assigns 3 to \( \text{Result} \). This result is stored for a later invocation since the function is a once function. Then, the recursive call is invoked with argument 2. But this invocation is not the first invocation, so the second invocation ignores the argument and returns the stored value (in this case 3). Thus, the result of invoking \( \text{factorial}(3) \) is \( 3 \ast 3 \). If we did not split the assignment, the result would be 0 because \( \text{factorial}(2) \) would return the result obtained so far which is the default value of \( \text{Result} \). This corresponds to a semantics where recursive calls are replaced by \( \text{Result} \).

To be able to develop a sound semantics for once functions, we need to consider all the possible cases described above. To fulfill this goal, we present a pseudo-code of once functions. Given a once function \( m \) with body \( b \), the pseudo-code is the following:
if not m_done then
  m_done := true
execute the body b
  if body triggers an exception then
    m_exc := true
  end
end
if m_exc then
  throw new exception
else
  Result := m_result
end

We assume the variables m_done, m_exc, and m_result are global variables, which exist one per routine implementation and can be shared by all invocations of that function. Furthermore, we assume the body of the function sets the result variable m_result whenever Result is assigned to. Now, we can see more clearly why the invocation of factorial (3) returns 9. In the first invocation, first the global variable m_done is set to false, and then the function’s body is executed. The second invocation returns the stored value 3 because m_done is false.

To define the semantics for once functions, we introduce global variables to store the information whether the function was invoked before or not, to store whether it triggers an exception or not, and to store its result. These variables are T@m_done, T@m_result, and T@m_exc. There is only one variable T@m_done per every routine implementation T@m. Descendants of the class T that do not redefine the routine m inherit the routine implementation T@m, therefore they share the same global variable T@m_done.

Given a once function m implemented in the class T, T@m_done returns true if the once function was executed before, otherwise it returns false; T@m_result returns the result of the first invocation of m; and T@m_exc returns true if the first invocation of m produced an exception, otherwise it returns false. Since the type of the exception is not used in the exception mechanism, we use a boolean variable T@m_exc, instead of a variable of type EXCEPTION. We omit the definition of a global initialization phase T@m_done = false, T@m_result = default value, and T@m_exc = false.

The invocation of a once function is defined in four rules (rules 3.11-3.14, Figure 3.7). Rule (3.11) describes the normal execution of the first invocation of a once function. Before its execution, the global variable T@m_done is set to true. Then, the function body is executed. We assume here that the body updates the variable T@m_result whenever it assigns to Result. Rule (3.12) models the first invocation of a once function that terminates with an exception. The function is executed and terminates in the state σ’. The result of the once function m is the state σ’ where the variable T@m_exc is set to true to express that an exception was triggered. In rule (3.13), the first invocation of the once function
3.2 A Logic for Eiffel

\[ T@m = impl(\tau(\sigma(y)), m) \quad T@m \text{ is a once routine} \]
\[
\sigma(T@m\_done) = false \\
\langle \sigma[T@m\_done := true, \text{Current} := y, p := \sigma(e)], \text{body}(T@m) \rangle \rightarrow \sigma', \text{normal} \\
\langle \sigma, x := y.S:m(e) \rangle \rightarrow \sigma'[x := \sigma'(\text{Result})], \text{normal} \quad (3.11) \]

\[ T@m = impl(\tau(\sigma(y)), m) \quad T@m \text{ is a once routine} \]
\[
\sigma(T@m\_done) = false \\
\langle \sigma[T@m\_done := true, \text{Current} := y, p := \sigma(e)], \text{body}(T@m) \rangle \rightarrow \sigma', \text{exc} \\
\langle \sigma, x := y.S:m(e) \rangle \rightarrow \sigma'[T@m\_exc := true], \text{exc} \quad (3.12) \]

\[ T@m = impl(\tau(\sigma(y)), m) \quad T@m \text{ is a once routine} \]
\[
\sigma(T@m\_done) = true \\
\sigma(T@m\_exc) = false \\
\langle \sigma, x := y.S:m(e) \rangle \rightarrow \sigma[x := \sigma(T@m\_result)], \text{normal} \quad (3.13) \]

\[ T@m = impl(\tau(\sigma(y)), m) \quad T@m \text{ is a once routine} \]
\[
\sigma(T@m\_done) = true \\
\sigma(T@m\_exc) = true \\
\langle \sigma, x := y.S:m(e) \rangle \rightarrow \sigma, \text{exc} \quad (3.14) \]

Fig. 3.7: Operational Semantics for Once Routines

terminates normally, the remaining invocations restore the stored value using the variable \( T@m\_result \). In rule 3.14, the first invocation of \( m \) terminates with an exception, so the subsequent invocations of \( m \) trigger an exception, too.

The previous rules are still valid for non-once routines. Since the assignment to \( \text{Result} \) in a once routine updates the variable \( T@m\_result \), we need to extend its semantics. Let \( T@m \) be the routine implementation where the assignment takes place, the semantics is defined as follows:

\[ \sigma(e) = \text{exc} \]
\[
\langle \sigma, \text{Result} := e \rangle \rightarrow \sigma, \text{exc} \quad (3.15) \]

\[ \sigma(e) \neq \text{exc} \quad T@m \text{ is a once routine} \]
\[
\langle \sigma, \text{Result} := e \rangle \rightarrow \sigma[\text{Result} := \sigma(e), T@m\_result := \sigma(e)], \text{normal} \quad (3.16) \]
Multiple Inheritance

The complications of multiple inheritance can be elegantly captured by a revised definition of the function $impl$. While $impl(T, m)$ traverses $T$’s parent classes, it can take redefinition, undefinition, and renaming into account. In particular, $impl$ is undefined for deferred routines (abstract methods) or when an inherited routine has been undefined. Figure 3.8 shows an example of inheritance using the features rename and redefine. Table 3.1 presents an example of the application of the function $impl$ using the class declarations of Figure 3.8.

```
class A
  feature m do ... end
end

class B
  feature m do ... end
end

class C
  inherit A
  B rename m as n end
end

class D
  inherit C redefine m end
  feature m do ... end
end

class E
  inherit C rename m as m2 end
end
```

Fig. 3.8: Example of Inheritance using Rename and Redefine.

<table>
<thead>
<tr>
<th>$impl(A, m)$</th>
<th>$impl(B, m)$</th>
<th>$impl(C, m)$</th>
<th>$impl(C, n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A@m$</td>
<td>$B@m$</td>
<td>$A@m$</td>
<td>$B@m$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$impl(D, m)$</th>
<th>$impl(D, u)$</th>
<th>$impl(E, m)$</th>
<th>$impl(E, m2)$</th>
<th>$impl(E, n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D@m$</td>
<td>$B@m$</td>
<td>undefined</td>
<td>$A@m$</td>
<td>$B@m$</td>
</tr>
</tbody>
</table>

Tab. 3.1: Example of the Application of the Function $impl$.  

Definition of the function $impl$. Following, we present the definition of the function $imp$. In particular, $impl$ is undefined for deferred routines (abstract methods) or when an inherited routine has been undefined.
Given a class declaration list \(\text{env}\) (the list of classes that defines the program), a type \(t\), and a routine \(r\), \(\text{impl}\) returns the routine implementation where the routine \(r\) is defined. To do it, it takes the class \(t\) and looks for the routine declaration in \(t\). If \(r\) is defined in \(t\) then it returns the routine implementation \(t@r\); otherwise it searches in all the ancestors of \(t\).

The \(\text{impl}\) function is defined as follows:

\[
\text{impl} : \text{ClassDeclaration list} \times \text{Type} \times \text{RoutineId} \to \text{RoutineImp}
\]
\[
\text{impl} \ \text{env} \ t \ rId = \begin{cases} 
\text{t@rId} & \text{if (defined t rId)} \\
(\text{implementation} \ \text{env} \ (\text{list_inherits} \ \text{env} \ t \ rId)) & \text{else}
\end{cases}
\]

The \(\text{impl}\) function is generalized using the function \(\text{implementation}\). This function takes a list of types and routines. The routine is used to handle renaming. Its definition is as follows:

\[
\text{implementation} : \text{ClassDeclaration list} \times (\text{(Type} \times \text{RoutineId}) \times \text{list}) \to \text{RoutineImp}
\]
\[
\text{implementation} \ \text{env} \ (t, \ rId)\#xs = \begin{cases} 
\text{impl} \ \text{env} \ t \ rId & \text{if (deep_defined} \ \text{env} \ t \ rId) \\
(\text{implementation} \ \text{env} \ xs) & \text{else}
\end{cases}
\]

The function \(\text{deep_defined}\) yields \(\text{true}\) if only if given a class declaration list \(\text{env}\), a type \(t\), and a routine \(r\), \(r\) is defined in \(t\) or in any of its ancestors classes. This function uses the auxiliary function \(\text{deep_defined_list}\) which takes a list of types and routines to handle redefinition. The definitions are as follows:

\[
\text{deep_defined} : \text{ClassDeclaration list} \times ((\text{Type} \times \text{RoutineId}) \times \text{list}) \to \text{Bool}
\]
\[
\text{deep_defined} \ \text{env} \ cDecl \ rId = \text{undefined}
\]
\[
\text{deep_defined} \ \text{env} \ cDecl \ rId = \begin{cases} 
\text{true} & \text{if (defined} \ cDecl \ rId) \\
(\text{deep_defined_list} \ \text{env} \ (\text{list_inherits} \ \text{env} \ cDecl \ rId)) & \text{else}
\end{cases}
\]

\[
\text{deep_defined_list} : \text{ClassDeclaration list} \times ((\text{Type} \times \text{RoutineId}) \times \text{list}) \to \text{Bool}
\]
\[
\text{deep_defined_list} \ \text{env} \ [] = \text{false}
\]
\[
\text{deep_defined_list} \ \text{env} \ (t, \ rId)\#xs = (\text{deep_defined} \ \text{env} \ t \ rId) \lor (\text{deep_defined_list} \ \text{env} \ xs)
\]

Given a type \(t\), and a routine \(r\), the function \(\text{list_inherits}\) yields a list of the parents classes and routines where the routine \(r\) might be defined. This function considers renaming and undefining of routines. Its definition is the following:
Given a list of class declarations \( \text{env} \), an inheritance clause \( \text{inh} \), and a routine \( r \), the function \( \text{listInh} \) yields a list of types and routines where the routine \( r \) might be defined. If the routine is undefined in the parent class, the function does not search its implementation. If the routine is renamed, it searches for the new routine name. This function is defined as follows:

\[
\text{list} \_ \text{inh} : \text{ClassDeclaration list} \times \text{Type} \times \text{RoutineId} \rightarrow (\text{Type} \times \text{RoutineId}) \text{ list}
\]

\[
\text{list} \_ \text{inh} \ [\ ] \ t \ rId = [\ ]
\]

\[
\text{list} \_ \text{inh} \ \text{env} \ t \ rId = (\text{list} \_ \text{inh} \ \text{env} \ (\text{parents} \ t) \ rId)
\]

where the function \( \text{is\_undefined} \) yields \( \text{true} \) if the routine is undefined in the inheritance clause, and the function \( \text{renamed\_type} \) yields the name of the routine considering renaming (if the routine \( r \) is not renamed, it yields the same routine \( r \)).

### 3.2.3 A Programming Logic for Eiffel

The logic for Eiffel is a Hoare-style logic; it is an extension of the logic for Mate (presented in Section 3.1.4). Properties of routines and routine bodies are also expressed by Hoare triples of the form \( \{P\} \ S \ {Q_n, Q_e} \), where \( P, Q_n, Q_e \) are formulas in first order logic, and \( S \) is a routine or an instruction. The third component of the triple consists of a normal postcondition \( (Q_n) \), and an exceptional postcondition \( (Q_e) \).

Since the subset of Eiffel includes exception handling using \text{rescue} \ clauses and \text{retry} \ variable, we have to extend the signatures of contracts. Preconditions and postconditions of Hoare triples are formulas over \( \Sigma \cup \{\text{Current}, p, \text{Result}, \text{Retry}\} \cup \text{Var}(r) \) where \( r \) is a routine, and \( \text{Var}(r) \) denotes the set of local variables of \( r \). In the case of Eiffel, \( \Sigma \) also contains the special variables for once routines. Note that we assume \( \text{Var}(r) \) does not include the \text{Result} variable and the \text{Retry} variable, it only includes the local variables declared by the programmer. Routine preconditions are formulas over \( \Sigma \cup \{\text{Current}, p, \$\} \), routine postconditions for normal termination are formulas over \( \Sigma \cup \{\text{Result}, \$\} \), and routine exceptional postconditions are formulas over \( \Sigma \cup \$ \).
3.2 A Logic for Eiffel

Exception Handling

The operational semantics presented in Section 3.2.2 shows that a rescue clause is a loop. The loop body $s_2; s_1$ iterates until no exception is thrown in $s_1$, or Retry is false. To be able to reason about this loop, we introduce an invariant $I_r$. We call this invariant rescue invariant. The rule is defined as follows:

$$
\frac{P \Rightarrow I_r}{A \vdash \{ I_r \} \ s_1 \ \{ Q_n , Q_e \}}
\frac{A \vdash \{ Q_e \} \ s_2 \ \{ (\text{Retry} \Rightarrow I_r) \land (\neg \text{Retry} \Rightarrow R_e) , R_e \}}{A \vdash \{ P \} \ s_1 \ \text{rescue} \ s_2 \ \{ Q_n , R_e \}}
$$

This rule is applied to any routine with a rescue clause. If the do clause, $s_1$, terminates normally then the rescue clause is not executed and the postcondition is $Q_n$. If $s_1$ triggers an exception, the rescue clause executes. If the rescue clause, $s_2$, terminates normally, and the Retry variable is true then control flow transfers back to the beginning of the routine and $I_r$ holds. If $s_2$ terminates normally and Retry is false, the routine triggers an exception and $R_e$ holds. If both $s_1$ and $s_2$ trigger an exception, the last one takes precedence, and $R_e$ holds.

Once Routines

To define the logic for once routines, we use the global variables $T@m\_done$, $T@m\_result$, and $T@m\_exc$, which store if the once routine was executed before or not, the result, and the exception. Let $P$ be the following precondition, where $T\_M\_RES$ is a logical variable:

$$
P \equiv \left\{ \begin{array}{l}
(\neg T@m\_done \land P') \lor \\
(T@m\_done \land P'' \land T@m\_result = T\_M\_RES \land \neg T@m\_exc) \lor \\
(T@m\_done \land P'' \land T@m\_exc)
\end{array} \right\}
$$

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

$$
Q'_n \equiv \left\{ \begin{array}{l}
T@m\_done \land (Q_n \lor (P'' \land \text{Result} = T\_M\_RES \land T@m\_result = T\_M\_RES))
\end{array} \right\}
$$

$$
Q'_e \equiv \left\{ \begin{array}{l}
T@m\_done \land (Q_e \lor P''')
\end{array} \right\}
$$

The rule for once functions is defined as follows:

$$
\frac{A, \{ P \} \ T@m \ \{ Q'_n , Q'_e \} \vdash \\
\{ P'[false/T@m\_done] \land T@m\_done \} \ \text{body}(T@m) \ \{ Q_n , Q_e \}}{A \vdash \{ P \} \ T@m \ \{ Q'_n , Q'_e \}}
$$
In the precondition of the body of $T@m$ (the hypothesis of the rule), $T@m\_done$ is true to model recursive call as illustrated in the example presented in Section 3.2.2. The replacement $P'[false/T@m\_done]$ is done because $P'$ can refer to $T@m\_done$. In the postcondition of the rule, under normal termination, either the function $T@m$ is executed and $Q_n$ holds, or the function is not executed since it was already executed, and $P''$ holds. In both cases, $T@m\_done$ is true and $T@m\_exc false$. In the case an exception is triggered, $Q_e \lor P'''$ holds.

To conclude the presentation of once routine, the following rule is defined to assign to Result in a once routine. Let $T@m$ be the routine implementation where the assignment takes place. The rule is:

$$
T@m is a once routine
\vdash \begin{cases}
(safe(e) \land P[e/Result, e/T@m\_result]) \lor \\
(\neg safe(e) \land Q_e)
\end{cases} \quad Result := e \quad \{ P, Q_e \}
$$

### 3.2.4 Example

In this section, we present an application of the logic for Eiffel. The function $safe\_division$, defined in the class $MATH$, implements an integer division which always terminates normally. If $y$ (the second operand) is zero, this function returns $x$ (the first operand); otherwise it returns the integer division $x//y$. This function is implemented in Eiffel using a rescue clause. If the division triggers an exception, this exception is handled by the rescue clause setting $z$ to 1 and retrying. The local variable $z$ is needed due to, in Eiffel, arguments cannot be assigned to. For simplicity, we assume the class $MATH$ has no descendants. The function is implemented as follows:

```eiffel
safe\_division (x,y: INTEGER): INTEGER
local
    z: INTEGER
do
    Result := x // (y+z)
ensure
    zero: y = 0 implies Result = x
    not\_zero: y /= 0 implies Result = x // y
rescue
    z := 1
    Retry := true
end
```

Applying the logic for Eiffel, we have proven that the routine $safe\_division$ satisfies the following specification:
3.2 A Logic for Eiffel

\[
\{ \text{true} \} \quad \text{MATH:safe\_division} \quad \{ \ Q \ , \ \text{false} \ \}
\]

where
\[
Q \equiv (y = 0 \Rightarrow Result = x) \land (y/ = 0 \Rightarrow Result = x/\ y)
\]

Figure 3.9 presents the proof for the body of the routine \textit{safe\_division}. To verify this, we first apply the rescue rule (presented on Section 3.2.3). The \textit{retry invariant} is:

\[
(y \neq 0 \land z = 0) \lor (y = 0 \land (z = 1 \lor z = 0))
\]

Finally, we verify the body of the do block and the body of the \texttt{rescue} clause using the assignment, and the compound rules. Thus, we prove:

\[
\text{safe\_division} \ (x, y: \ \text{INTEGER}): \ \text{INTEGER}
\]

\[
\text{local} \quad z: \ \text{INTEGER}
\]

\[
do \quad \{ (y \neq 0 \land z = 0) \lor (y = 0 \land (z = 1 \lor z = 0)) \}
\]

\[
\text{Result} := x /\ (y+z)
\]

\[
\{ (y = 0 \Rightarrow Result = x) \quad \land \quad (y \neq 0 \Rightarrow Result = x/\ y) \}, (y = 0 \land z = 0)\}
\]

\[
\text{ensure} \quad \text{zero: } y = 0 \ \text{implies Result} = x
\]

\[
\text{not\_zero: } y /\ = 0 \ \text{implies Result} = x /\ y
\]

\[
\text{rescue} \quad \{ y = 0 \land z = 0 \}
\]

\[
z := 1
\]

\[
\{ (y = 0 \land z = 1), \ \text{false} \}
\]

\[
\text{Retry} := \text{true}
\]

\[
\{ (\text{Retry} \land (y = 0 \land z = 1)) , \ \text{false} \}
\]

\]

Fig. 3.9: Example of an Eiffel Source Proof.

\[
\{ \text{true} \} \quad \text{body(MATH@safe\_division)} \quad \{ \ Q \ , \ \text{false} \ \}
\]

Similar to the example of the function \textit{sum} (presented on Section 3.1.5), we can show that the routine implementation \textit{MATH@safe\_division} satisfies its specification applying the routine implementation rule:
Since safe\_division is implemented in the class MATH, we have:

\[
\text{impl}(MATH, \text{safe\_division}) = MATH@safe\_division
\]

Finally, we can apply the class rule, and we prove:

\[
\begin{align}
\{ \tau(\text{Current}) \prec MATH \land true \} & \quad \text{MATH:safe\_division} & \{ Q, false \} \\
\{ \tau(\text{Current}) = MATH \land true \} & \quad \text{impl}(MATH, \text{safe\_division}) & \{ Q, false \}
\end{align}
\]

\[
\{ \tau(\text{Current}) \preceq MATH \land true \} \quad \text{MATH:safe\_division} \quad \{ Q, false \}
\]

### 3.3 A Logic for Java-like Programs

In the previous section, we have presented a logic for Eiffel. The subset of Eiffel includes exception handling, multiple inheritance, and once routines. In this section, we extend the Mate language to a subset of Java.

#### 3.3.1 The Java Language

Figure [3.10](#) presents the syntax of the subset of Java. This subset supports while and break instructions, and try-catch, try-finally, and throw instructions. To avoid return instructions, we assume that the return value of every method is assigned to a special local variable named Result. These are the only discordance with respect to Java.

#### 3.3.2 Operational Semantics

To define the operational semantics for the subset of Java, we use the same memory model described in Section [3.1.2](#). Furthermore, the program state is the same as the Mate program state. To model break instructions, we extend definition of the transitions of the operational semantics. These transitions have the form:

\[
\langle \sigma, S \rangle \rightarrow \sigma', \chi
\]

where \( \sigma \) and \( \sigma' \) are states, \( S \) is an instruction, and \( \chi \) is the current status of the program. The value of \( \chi \) can be either the constant normal, or \( \text{exc} \), or break. The transition \( \langle \sigma, S \rangle \rightarrow \sigma', \text{normal} \) expresses that executing the instruction \( S \) in the state \( \sigma \) terminates normally in the state \( \sigma' \). The transition \( \langle \sigma, S \rangle \rightarrow \sigma', \text{exc} \) expresses that executing the instruction \( S \) in the state \( \sigma \) terminates with an exception in the state \( \sigma' \). The transition
3.3 A Logic for Java-like Programs

Program ::= ClassDecl*
ClassDecl ::= class ClassId [: Type] ClassBody
Type ::= BoolT | IntT | ClassId | VoidT
ClassBody ::= MemDecl*
MemDecl ::= Type AttributeId
| Routine
Routine ::= Type RoutineId (Type)
            { Instr }
Instr ::= VarId := ExpE
         | Instr; Instr
         | if BoolExp then Instr else Instr end
         | VarId := new Type()
         | VarId := VarId.Type.getAttributeId
         | VarId.Type.getAttributeId := Exp
         | VarId := VarId.Type : RoutineId (Exp )
         | while (BoolExp) Instr
         | break
         | try Instr catch (Type VarId) Instr
         | try Instr finally Instr
         | throw Exp
         | assert Exp
Instr, Exp,
ExpE, BoolExp ::= //as defined in Figure 3.1

Fig. 3.10: Syntax of the Subset of Java.

\( \langle \sigma, S \rangle \rightarrow \sigma', break \) expresses that executing the instruction \( S \) in the state \( \sigma \) terminates with the execution of a break instruction in the state \( \sigma' \).

While loops and break instructions

In this section, we define the operational semantics for while and break instructions. Since the transition \( \langle \sigma, S \rangle \rightarrow \sigma', \chi \) introduces a new value for \( \chi \), we need to extend the operational semantics for Mate to Java. Here, we extend this transition for compound, the remaining instructions are similar. Figure 3.11 shows the operational semantics for compound, break, and while instructions.

**Compound.** Compound is defined by three rules: in rule (3.11.1) the instruction \( s_1 \) is executed and an exception is triggered. The instruction \( s_2 \) is not executed, and the state of the compound is the state produced by \( s_1 \). In rule (3.11.2), the instruction \( s_1 \) is executed and a break instruction is executed. The instruction \( s_2 \) is not executed, and the state of the compound is the state produced by \( s_1 \). In rule (3.11.3), \( s_1 \) is executed and terminates normally. The state of the compound is the state produced by \( s_2 \).
Break Instruction. The break instruction changes the program status \( \chi \) to break without modifying the state \( \sigma \), rule (3.11.4).

While Instruction. In rule (3.11.5), since the condition of the while evaluates to false, then the body of the loop is not executed producing the state \( \sigma \). If the condition of the while is true, in rule (3.11.6), the instruction \( s_1 \) is executed, but it triggers an exception. Thus, the state of the loop is \( \sigma' \), and the status is exc. In rule (3.11.7), the instruction \( s_1 \) executes a break instruction, then the while terminates in the state \( \sigma'' \); the program status is normal. Finally, in rule (3.11.8), \( s_1 \) terminates normally and the condition is evaluated to false, the returned state is the one produced by the new execution of the loop.

![Fig. 3.11: Operational Semantics for Compound, while, and break Instructions in Java](image)
### 3.3 A Logic for Java-like Programs

**Throw Instruction**

\[ \langle \sigma, \text{throw } e \rangle \rightarrow \sigma[\text{excV} := e], \text{exc} \quad (3.12\,1) \]

**Try-Catch Instruction**

- Rule (3.12.2): Normal termination
  \[ \langle \sigma, \text{try } s_1 \text{ catch } (T \ e) \ s_2 \rangle \rightarrow \sigma', \text{normal} \]

- Rule (3.12.3): Exception triggered with \( \tau'(\text{excV}) \nless T \)
  \[ \langle \sigma, \text{try } s_1 \text{ catch } (T \ e) \ s_2 \rangle \rightarrow \sigma', \text{exc} \]

- Rule (3.12.4): Exception caught by \( \text{catch} \) block
  \[ \langle \sigma, \text{try } s_1 \text{ catch } (T \ e) \ s_2 \rangle \rightarrow \sigma'', \chi \]

Fig. 3.12: Operational Semantics for throw and try-catch Instructions

To define the semantics of try-catch and throw, we use a variable excV that stores the current exception. Figure 3.12 presents the operational semantics for try-catch and throw instructions.

In the semantics of the throw instruction, rule (3.12.1), the exception value excV is assigned with the expression \( e \), and the instruction terminates in the status exc. The semantics for try-catch is defined by three rules. In rule (3.12.2) the instruction \( s_1 \) terminates normally, so the instruction \( s_2 \) is not executed, and the state of the try-catch is the state produced by \( s_1 \). In rule (3.12.3), the instruction \( s_1 \) is executed and triggers an exception. Since the dynamic type of the exception is not a subtype of \( T \) (expresses as \( \tau(\text{excV}) \nless T \)), the exception is propagated, and the state of the try-catch is the state produced by \( s_1 \). In rule (3.12.4), the instruction \( s_1 \) triggers an exception and the exception is caught by the catch block. The state of the try-catch is the state produced by \( s_2 \).

**Semantics for try-finally Instructions.** The combination of while, break, try-catch, and try-finally instructions produces an interesting subset of Java, especially from the point of view of the semantics of try-finally instructions. The following example illustrates a special case that we have to treat:
void foo (int b) {
    b = 1;
    while (true) {
        try {
            throw new Exception();
        }
        finally {
            b++; // This line ensures that the loop terminates
            break;
        }
    }
    b++; // This line is redundant but included for completeness
}

In the foo routine, an exception is triggered in the try block but it is never caught. However, the loop terminates normally due to the execution of the break instruction in the finally block transfers control to the end of the loop. Thus, the value of b at the end of the function foo is 3. If an exception occurs in a try block, it will be re-raised after the execution of the finally block. If both the try and the finally block throw an exception, the latter takes precedence. The following table summarizes the status of the program after the execution of the try-finally (exc_1 denotes the exception triggered by the instruction s_1, and exc_2 the exception triggered by s_2):

<table>
<thead>
<tr>
<th>finally</th>
<th>normal</th>
<th>break</th>
<th>exc_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>try</td>
<td>normal</td>
<td>break</td>
<td>exc_2</td>
</tr>
<tr>
<td>break</td>
<td>break</td>
<td>break</td>
<td>exc_2</td>
</tr>
<tr>
<td>exc_1</td>
<td>exc_1</td>
<td>break</td>
<td>exc_2</td>
</tr>
</tbody>
</table>

The operational semantics for try-finally instructions is defined as follows:

\[
\langle \sigma, s_1 \rangle \rightarrow \sigma', \chi \quad \langle \sigma', s_2 \rangle \rightarrow \sigma'', \text{normal} \\
\quad \quad \rightarrow \langle \sigma, \text{try } s_1 \text{ finally } s_2 \rangle \rightarrow \sigma'', \chi \quad \text{(3.17)}
\]

\[
\langle \sigma, s_1 \rangle \rightarrow \sigma', \chi \quad \langle \sigma', s_2 \rangle \rightarrow \sigma'', \text{break} \\
\quad \quad \rightarrow \langle \sigma, \text{try } s_1 \text{ finally } s_2 \rangle \rightarrow \sigma'', \text{break} \quad \text{(3.18)}
\]

\[
\langle \sigma, s_1 \rangle \rightarrow \sigma', \chi \quad \langle \sigma', s_2 \rangle \rightarrow \sigma'', \text{exc} \\
\quad \quad \rightarrow \langle \sigma, \text{try } s_1 \text{ finally } s_2 \rangle \rightarrow \sigma'', \text{exc} \quad \text{(3.19)}
\]

In these rules, the state is produced by the execution of the instruction s_2 starting with the state produced by s_1; the status of the program changes in each case. In rule [3.17], if the instruction s_2 terminates normally, the status of the try-finally is the status produced by s_1. In rule [3.18], the instruction s_2 terminates with a break, and the status of the try-finally is break. Similarly, in rule [3.19] s_2 terminates with an exception and the status of the try-finally is exc.
A Programming Logic for Java

The logic for Java is a Hoare-style logic extended from the logic for Mate (presented in Section 3.1.4). Since instructions can execute break instructions, properties of methods and properties of instructions are expressed using different Hoare triples.

Properties of methods are expressed by Hoare triples of the form \( \{ P \} \ m \ \{ Q_n, Q_e \} \), where \( P, Q_n, Q_e \) are first-order formulas and \( m \) is a virtual method \( T: m \), or a method implementation \( T@m \). The third component of the triple consists of a normal postcondition \( Q_n \), and an exceptional postcondition \( Q_e \). We call such a triple method specification.

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

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

Rules

Figure 3.13 shows the rules for compositional, while, break, try-catch, and throw instructions. In the compositional rule, the instruction \( s_1 \) is executed first. The instruction \( s_2 \) is executed if and only if \( s_1 \) terminates normally. In the while rule, the execution of the instruction \( s_1 \) can produce three results: either (1) \( s_1 \) terminates normally and \( I \) holds, or (2) \( s_1 \) executes a break instruction and \( Q_b \) holds, or (3) \( s_1 \) throws an exception and \( R_e \) holds. The postcondition of the while instruction expresses that either the loop terminates normally and \( (I \land \neg e) \lor Q_b \) holds or throws an exception and \( R_e \) holds. The break postcondition is false, because after a break within the loop, execution continues normally after the loop.

The break axiom sets the normal and exception postcondition to false and the break postcondition to \( P \) due to the execution of a break instruction. Similar to break, the throw axiom modifies the postcondition \( P \) by updating the exception component of the state with the just evaluated reference.

In the try-catch rule, the execution of the instruction \( s_1 \) can produce three different results: (1) \( s_1 \) terminates normally and \( Q_n \) holds or terminates with a break and \( Q_b \) holds. In these cases, the instruction \( s_2 \) is not executed and the postcondition of the try-catch is the postcondition of \( s_1 \); (2) \( s_1 \) throws an exception and the exception is not caught. The instruction \( s_2 \) is not executed and the try-catch finishes in an exception mode. The
A Sound and Complete Logic for Object-Oriented Programs

Compositional Rule:
\[
\begin{align*}
\{ P \} & \quad s_1 \quad \{ Q_n, R_b, R_e \} \\
\{ Q_n \} & \quad s_2 \quad \{ R_n, R_b, R_e \}
\end{align*}
\]

While Rule:
\[
\begin{align*}
\{ e \land I \} & \quad s_1 \quad \{ I, Q_b, R_e \} \\
\{ I \} & \quad \text{while}(e) \quad s_1 \quad \{ (I \land \neg e) \lor Q_b), \text{false}, R_e \}
\end{align*}
\]

Break Axiom:
\[
\begin{align*}
\{ P \} & \quad \text{break} \quad \{ \text{false}, P, \text{false} \}
\end{align*}
\]

Throw Axiom:
\[
\begin{align*}
\{ P[e/excV] \} & \quad \text{throw} \quad e \quad \{ \text{false}, \text{false}, P \}
\end{align*}
\]

try-catch Rule:
\[
\begin{align*}
\{ P \} & \quad s_1 \quad \{ Q_n, Q_b, Q \} \\
\{ Q'_e[e/excV] \} & \quad s_2 \quad \{ Q_n, Q_b, R_e \}
\end{align*}
\]

\[
\begin{align*}
\{ P \} & \quad \text{try} \quad s_1 \quad \text{catch}(T \ e) \quad s_2 \quad \{ Q_n, Q_b, R \}
\end{align*}
\]

where
\[
Q \equiv ( (Q''_n \land \tau(excV) \not\leq T) \lor (Q'_e \land \tau(excV) \leq T) )
\]
\[
R \equiv (R_e \lor (Q''_e \land \tau(excV) \not\leq T) )
\]

try-finally Rule:
\[
\begin{align*}
\{ P \} & \quad s_1 \quad \{ Q_n, Q_b, Q_e \} \\
\{ Q \} & \quad s_2 \quad \{ R, R'_b, R'_e \}
\end{align*}
\]

\[
\begin{align*}
\{ P \} & \quad \text{try} \quad s_1 \quad \text{finally} \quad s_2 \quad \{ R'_n, R'_b, R'_e \}
\end{align*}
\]

where
\[
Q \equiv \left( (Q_n \land XTmp = \text{normal}) \lor (Q_b \land XTmp = \text{break}) \lor (Q_e[eTmp/excV] \land XTmp = \text{exc} \land eTmp = excV ) \right)
\]
\[
R \equiv \left( (R'_n \land XTmp = \text{normal}) \lor (R'_b \land XTmp = \text{break}) \lor (R'_e[eTmp/excV] \land XTmp = \text{exc}) \right)
\]

Fig. 3.13: Rules for Composition, while, break, try-catch, and throw Instructions.
postcondition is \( Q''_e \land \tau(excV) \not\leq T \), where \( \tau \) yields the runtime type of an object, \( excV \) is a variable that stores the current exception, and \( \leq \) denotes subtyping; (3) \( s_1 \) throws an exception and the exception is caught. In the postcondition of \( s_1 \), \( Q'_e \land \tau(excV) \leq T \) specifies that the exception is caught. Finally, \( s_2 \) is executed producing the postcondition. Note that the postcondition is not only a normal postcondition: it also has to take into account that \( s_2 \) can throw an exception or can execute a \textbf{break}.

To define the rule for \texttt{try-finally}, we use the fresh variable \( eTmp \) to store the exception occurred in \( s_1 \). This variable is needed because another exception might be raised and caught in \( s_2 \), and one still needs to have access to the first exception of \( s_1 \). To model all possible executions of a \texttt{try-finally} (as described in Section 3.3.2), we use the fresh variable \( \mathcal{X}Tmp \), which stores the status of the program after the execution of \( s_1 \). The possible values of \( \mathcal{X}Tmp \) are: \texttt{normal}, \texttt{break}, and \texttt{exc}. Depending on the status after the execution of \( s_2 \), one needs to propagate an exception or change the status of the program to \texttt{break}.

### 3.3.4 Example

Figure 3.14 presents an example of a proof of a Java function using \texttt{while}, \texttt{break}, and \texttt{try-finally} instructions. In the \texttt{foo} function, an exception is thrown in the \texttt{try} block with precondition \( b = 1 \). The \texttt{finally} block increases \( b \) and then executes a \texttt{break} instruction changing the status of the program to break mode (the postcondition is \( b = 2 \)). Using the logic for Java, we have shown that the routine \texttt{foo} satisfies the following specification:

\[
\{ \text{true} \} \quad \text{MATH:foo} \quad \{ \ b = 3 \ , \ false \ \}
\]

To prove that the body of the routine \texttt{foo} satisfies the following specification:

\[
\{ \text{true} \} \quad \text{body(MATH@foo)} \quad \{ \ b = 3, \text{false, false} \ \}
\]

we apply the \texttt{try-finally}, \texttt{throw}, \texttt{break} rules, and the assignment and compound rules. The \texttt{throw} instruction changes the precondition \( b = 1 \) to an exceptional postcondition. Using the \texttt{try-finally} rule, we can prove that postcondition of the \texttt{try-finally} is \{false, \( b = 2 \), false\}. To prove the loop, we use the break condition \( b = 2 \) and the invariant \texttt{false}. Thus, we obtain the postcondition \{\( b = 2, \text{false, false} \)\}. Figure 3.14 presents a sketch of this proof.

Applying the routine implementation rule, we can connect the specification of the body of \texttt{foo} and the method specification. The body of \texttt{foo} uses three postconditions to express the condition after a break, in the method specification, we only need two postconditions (the normal and the exceptional postcondition). Thus, we show:

\[
\begin{align*}
\{ \text{true} \} \quad \text{body(MATH@foo)} \quad \{ \ b = 3, \text{false, false} \ \} \\
\{ \text{true} \} \quad \text{MATH@foo} \quad \{ \ b = 3 \ , \ false \ \}
\end{align*}
\]
void foo (int b) {
    
    b = 1;
    
    while (true) {
        try {
            throw new Exception();
        } finally {
            
            b = b + 1;
            break;
        }
        
        
    }

    
    b = b + 1;
    
    b = b + 1;
    
    
}

Fig. 3.14: Example of a Proof of a Java Function using while, break, and try–finally Instructions.
3.4 Soundness and Completeness

The rest of the proof is similar to the proofs presented in Section 3.1.5 and Section 3.2.4. Since the class $MATH$ does not have descendent, we can prove:

\[ \tau(CURRENT) \prec MATH \land n > 1 \Rightarrow false \]

Then, applying the strength rule we show:

\[
\begin{array}{c}
\{ \text{false} \} MATH:foo \{ b = 3, \text{false} \} \\
\{ \tau(CURRENT) \prec MATH \land true \} MATH:foo \{ b = 3, \text{false} \}
\end{array}
\]

Since $foo$ is implemented in the class $MATH$, we have $impl(MATH, foo)=MATH@foo$. Therefore, we can conclude the proof by applying the class rule (page 29) using the above two proofs:

\[
\begin{array}{c}
\{ \tau(CURRENT) \prec MATH \land true \} MATH:foo \{ b = 3, \text{false} \} \\
\{ \tau(CURRENT) = MATH \land true \} impl(MATH, foo) \{ b = 3, \text{false} \}
\end{array}
\]

\[
\begin{array}{c}
\{ \tau(CURRENT) \preceq MATH \land true \} MATH:foo \{ b = 3, \text{false} \}
\end{array}
\]

3.4 Soundness and Completeness

We have proved soundness and completeness of the logic. The soundness proof runs by induction on the structure of the derivation tree for $\{ P \} s \{ Q_n , Q_e \}$. The completeness proof runs by induction on the structure of the instruction $s$ using a sequent which contains a complete specification for every routine implementation $T@m$. In this section, we state the theorems. The proofs are presented in Appendix B.

**Definition 3.** The triple $\models \{ P \} s \{ Q_n , Q_e \}$ if and only if:

for all $\sigma \models P : \langle \sigma, s \rangle \rightarrow \sigma', \chi$ then

- $\chi = normal \Rightarrow \sigma' \models Q_n$, and
- $\chi = exc \Rightarrow \sigma' \models Q_e$

**Theorem 1** (Soundness Theorem).

$\vdash \{ P \} s \{ Q_n , Q_e \} \Rightarrow \models \{ P \} s \{ Q_n , Q_e \}$

**Theorem 2** (Completeness Theorem).

$\models \{ P \} s \{ Q_n , Q_e \} \Rightarrow \vdash \{ P \} s \{ Q_n , Q_e \}$
3.5 Related Work

Abadi and Leino [11] have developed a logic to verify object-oriented programs. The language is a simple object-oriented language that handles objects with fields and methods. The operational semantics allows aliasing. They have proved soundness of the logic with respect to the operational semantics. Poetzsch-Heffter [105] describes how to integrate Larch-style specifications with Hoare-style logics. The technique is applied to object-oriented programs supporting subtyping. Hensel et al. [46] develop a formal language to describe specifications of classes in object-oriented languages. The source language includes object-oriented features such as classes with inheritance, and overriding. They briefly describe how to extend their system to multiple inheritance. The languages used in those works are simpler than the languages we use in this chapter; their works do not handle exception handling. Furthermore, our work not only proves soundness of the logic with respect to the operational semantics but it also proves completeness.

More recent works [57, 52, 124] have extended logics to cover exception handling and abrupt termination. For example, Huisman and Jacobs [52, 53] have developed a Hoare-style logic with abrupt termination. It includes not only exception handling but also while loops which may contain exceptions, breaks, continues, returns, and side-effects. The logic is formulated in a general type theoretical language and not in a specific language such as PVS or Isabelle. Oheimb [123, 124] has developed a Hoare-style calculus for a subset of JavaCard. The language includes side-effecting expressions, mutual recursion, dynamic method binding, full exception handling, and static class initialization. Oheimb has proven soundness and completeness of the logic. These logics formalize a Java-like exception handling which is different to the Eiffel exception handling presented in this chapter. Our work includes try-finally instructions, and it shows interesting aspects of the Java semantics which are not covered in Huisman and Jacobs’ work [52], and Oheimb’s work [123, 124]. This chapter also covers other object-oriented features such as the Eiffel once routines and multiple inheritance.

Schirmer [115, 116] defines a programming model for sequential imperative programs in Isabelle/HOL. The source language handles abrupt termination, side-effect expressions, and dynamic procedure calls. However, try-finally instructions are not discussed in that work. We extend the logic for dynamic procedure calls (function objects) in Chapter 4. Nipkow [89] presents a big and small step operational semantics for a Java-like language. The semantics handles try-catch and throw instructions. The big and small operational semantics are shown equivalent; the proof is formalized in Isabelle/HOL.

Logics such as separation logic [111, 97, 17], dynamic frames [59, 119, 120], and regions [1] have been proposed to solve a key issue for reasoning about imperative programs: framing. Separation logic has been adapted to verify object-oriented programs [101, 102, 36]; Parkinson [100] has extended separation logic to reason about Java programs; Parkinson and Bieman [101, 102] introduce abstract predicates: a powerful means to abstract from implementation details and to support information hiding and inheritance. Distefano
and Parkinson [36] develop a tool to verify Java programs based on the ideas of abstract predicates. Our work does not address how to structure the heap, however, it covers a wider range of features of object-oriented languages.

Logics have been also developed for bytecode languages. Bannwart and Müller [6] have developed a Hoare-style logic for a bytecode similar to Java Bytecode and CIL. The logic is based on Poetzsch-Heffter and Müller’s logic [107, 108], and it supports object-oriented features such as inheritance and dynamic binding. However, the bytecode logic does not include exception handling; so it does not include the CIL instructions .try catch and .try .finally. The Mobius project [76] has also developed a program logic for bytecode. The logic has been proved sound with respect the operational semantics, and it has been formalized in Coq. Other works [70, 110, 69] have formalized Java Bytecode programs. For example, Liu and Moore [69] have defined a deep embedding of Java Bytecode in ACL2; Quigley [110] formalizes Java Bytecode in Isabelle; and Luo et al. [70] extend separation logic to Java Bytecode.

With the goal of verifying bytecode programs, Pavlova [103] has developed an operational semantics, and a verification condition generator (VC) for Java Bytecode. Furthermore, she has shown the equivalence between the verification condition generated from the source program and the one generated from the bytecode.

An operational semantics and a verification methodology for Eiffel has been presented by Schöller [117]. The methodology uses dynamic frame contracts to be able to address the frame problem, and applies to a substantial subset of Eiffel. However, Schöller’s work only presents an operational semantics, and it does not include exceptions. Paige and Ostroff [99] define a refinement calculus for a subset of Eiffel. The subset of Eiffel includes simple instructions, but it does not handle attributes, exception handling, once routines, and multiple inheritance.

Our logic is based on Poetzsch-Heffter and Müller’s work [107, 108, 79], which we extended by new rules for Eiffel instructions and the Java instructions try-catch, try-finally, and break. This chapter is based on our earlier work [93, 80, 94].

3.6 Lessons Learned

We have presented a sound and complete logic for object-oriented programs. Here we report on some lessons on programming language design learned in the process.

Abrupt Termination

During the development of this work, we have formalized the current Eiffel exception handling mechanism. In the current version of Eiffel, retry is an instruction that can only be used in a rescue clause. When retry is executed, control is transferred to the beginning of the routine. If the execution of the rescue clause finishes without invoking
A Sound and Complete Logic for Object-Oriented Programs

a retry, an exception is triggered. Developing a logic for the current Eiffel would require
the addition of a third postcondition, to model the execution of retry (since retry is
another way of transferring control flow). Thus, we would use Hoare triples of the form
\{ P \} s \{ Q_n, Q_r, Q_e \} where s is an instruction, Q_n is the postcondition under
normal termination, Q_r the postcondition after the execution of a retry, and Q_e the
exceptional postcondition.

Such a formalization would make verification harder than the formalization we use
in this chapter, because the extra postcondition required by the retry instruction would
have to be carried throughout the whole reasoning. In this chapter, we have observed that
a rescue clause behaves as a loop that iterates until no exception is triggered, and that
retry can be modeled simply as a variable which guards the loop. Since the retry instruction
transfers control flow to the beginning of the routine, a retry instruction has a similar
behavior to a continue in Java or C#. Our proposed change of the retry instruction to
a variable will be introduced in the next revision of the language standard [75].

Since Eiffel does not have return instructions, nor continue, nor break instructions,
Eiffel programs can be verified using Hoare triples with only two postconditions. To model
object-oriented programs with abrupt termination in languages such as Java or C#, one
needs to introduce extra postconditions for return, break or continue (or we could
introduce a variable to model abrupt termination). If we wanted to model the current
version of Java, for example, we would also need to add postconditions for labelled breaks
and labelled continues. Thus, one would need to add as many postconditions as there are
labels in the program. These features for abrupt termination make the logic more complex
and harder to use.

Another difference between Eiffel and Java and C# is that Eiffel supports exceptions
using rescue clauses, and Java and C# using try-catch and try-finally instructions.
The use of try-finally makes the logic harder. The combination of try-finally and
break instructions makes the rules more complex and harder to apply because one has to
consider all possible cases in which the instructions can terminate (normal, break, return,
exception, etc).

However, we cannot conclude that the Eiffel’s exception handling mechanism is always
simpler for verification; although it eliminates the problems produced by try-finally,
break, and return instructions. Since the rescue clause is a loop, one needs a retry
invariant. When the program is simple, and it does not trigger many different exceptions,
defining this retry invariant is simple. But, if the program triggers different kinds of
exception at different locations, finding this invariant can be more complicated. Note
that finding this retry invariant is more complicated than finding a loop invariant since
in a loop invariant one has to consider only normal termination (and in Java and C#,
also continue instructions), but in retry invariants one needs to consider all possible
executions and all possible exceptions.
Multiple Inheritance

Introducing multiple inheritance to a programming language is not an easy task. The type system has to be extended, and this extension is complex. However, since the resolution of a routine name can be done syntactically, extending Poetzsch-Heffter and Müller’s logic [108] to handle multiple inheritance was not a complicated task. The logic was easily extended by giving a new definition of the function `impl`. This function returns the body of a routine by searching the definition in the parent classes, and considering the clauses redefine, undefine, and rename. The experience with this chapter indicates that the complexity of a logic for multiple inheritance is similar to a logic for single inheritance.

Once Routines

To verify once routines, we introduce global variables to express whether the once routine has been executed before or not, and whether the routine triggered an exception or not. With the current mechanism, the use of recursion in once functions does not increase the expressivity of the language. In fact, every recursive call can be equivalently replaced by `Result`. However, the rule for once functions is more complicated than it could be if recursion were omitted.

Recursive once functions would be more interesting if we changed the semantics of once routines. Instead of setting the global variable `done` before the execution of the body of the once function, we could set it after the invocation. Then the recursive once function would be invoked until the last recursive call finishes. Thus, for example, the result of the first invocation of `factorial(n)` would be `n!` (the function `factorial` is presented in Section 3.2.2). Later invocations of `factorial` would return the stored result. This change would work only for single recursion, and not for double recursions such as `f(x) = f(x - 1) + f(x - 2)`. However, this change would not simplify the logic, and we would need to use global variables to mark whether the once function was invoked before or not.

Analyzing the EiffelBase libraries, and the source code of the EiffelStudio compiler, we found that the predominant use of once functions is without arguments, which makes sense because arguments of subsequent calls are meaningless. Even though our rules for once functions are not overly complicated, verification of once functions is cumbersome because one has to carry around the outcome of the first invocation in proofs. It is unclear whether this is any simpler than reasoning about static methods and fields [66].
Chapter 4

Reasoning about Function Objects

Object-oriented design makes a clear choice in dealing with the basic duality between objects and operations (data and functions): it bases system architecture on the object, more precisely the object types as represented by classes, and attaching any operation to one such class. Functional programming languages, on the other hand, use functions as the primary compositional elements. The two paradigms are increasingly borrowing from each other: functional programming languages such as OCaml integrate object-oriented ideas, and a number of object-oriented languages now offer a mechanism to package operations (routines, methods) as objects. In the dynamically typed world, the idea goes back at least to Smalltalk with its blocks; among statically typed languages, C# has introduced delegates, Eiffel agents, and Scala function objects.

The concept of agent or delegate is, in its basic form, very simple, with immediate applications. A typical one, in a Graphical User Interface system, is for some part of a system to express its wish to observe (in the sense of the Observer pattern [40]) events of a certain type, by registering a procedure to be executed in response:

\[
\text{US\_map.\text{left\_click}.subscribe (agent show\_state\_votes)}
\]

This indicates that whenever a left\_click event occurs on the map, the given procedure show\_state\_votes should be executed. The routine subscribe takes as argument an agent representing a procedure. Since the agent is a formal argument, subscribe does not know which exact procedure, such as show\_state\_votes, it might represent; but it can call it all the same, through a general procedure call applicable to any agent, and any target and argument object.

Agents (we will stay with this term but much of the discussion applies to other language variants) appear in such examples as a form of function pointers as available for example in C and C++. But they go beyond this first analogy. First, they are declared with a signature and hence provide a statically typed mechanism, whereas a function pointer just denotes whatever is to be found in the corresponding memory address. Next, an agent represents a true routine abstraction with an operation to call the underlying routine.
These mechanisms have proved attractive to object-oriented programmers but they also raise new verification challenges: how do we prove programs taking advantage of them?

In the previous chapter, we have presented a semantics for a subset of Eiffel and Java. The semantics for Eiffel handles most of Eiffel’s main features, however, it omits agents. This chapter extends the Eiffel subset, and introduces a specification and verification technique for agents. Our approach uses side effect free (pure) routines to specify the pre- and postcondition of agents. To specify routines that take agents as arguments, we use these pure routines. The basic idea, developed in the following sections, is that to prove a property of an agent call, \( a.call(t, \text{arg}) \), it suffices to prove that the precondition of the agent \( a \) holds before the invocation, and then we can assume that the postcondition of \( a \) holds. Although we focus on Eiffel agents, it should be possible to apply the results to mechanisms addressing similar goals in other languages, in particular, C# delegates.

To show the practicability of our verification methodology, we have developed an automatic verifier for agents. Our verifier follows the architecture of the Spec# verifier [8]. Given an Eiffel program, the tool generates a BoogiePL file, and then it uses the Boogie [8] verifier to prove the generated file. Using previous work on pure routines [33, 67], we have encoded the pre and postconditions of agents as mathematical functions, which yields the value of the agent pre- and postcondition.

The four main technical contributions of this chapter are: (1) the idea of using pure routines to model abstractly the agent pre- and postcondition; (2) a specification and verification methodology for function objects; (3) an automatic verification tool; and (4) the demonstration of the approach’s practicality through a set of proofs, of a sequence of examples of increasing difficulty, including one previously described as an open problem, and more agent-intensive programs which implement graphical user interfaces.

The chapter is organized as follows: Section 4.1 describes the agent mechanism; Section 4.2 shows example applications of agents and their verification challenge. Section 4.3 presents the verification method. The development of the automatic verifier for agents is described in Section 4.4. Section 4.5 discusses related work.

This chapter is based on the published works [91, 92].

4.1 The Agent Mechanism

Agents represent a routine abstraction. The type of an agent expression is based on one of four library classes with an inheritance structure shown in Figure 4.1. FUNCTION (if the underlying routine is a function, returning a result), PREDICATE (for a function returning a boolean result) or PROCEDURE. These classes all descend from ROUTINE which introduces the procedure call to call the associated routine with a tuple argument.

\footnote{To simplify the notation, we use a slight variant of the Eiffel.}
4.1 The Agent Mechanism

For a FUNCTION or PREDICATE agent the result of the last call is available through
the routine last_result, or one can use fa.item([a, ...]) which calls fa and returns the result.
The agent classes are generic; the type of agent f, built on a function f(a : T) : V from
a class C, is FUNCTION [C, TUPLE [T], V]. The second generic parameter is a tuple
type, representing the open arguments.

The agent mechanism provides flexibility, in particular by allowing open arguments,
as in agent g(? , ? , z) where g has three arguments but the agent has only two, to be
provided to call (the question marks indicate where these open arguments lie). The
basic syntax agent g is simply an abbreviation for keeping all arguments open, as in
the more explicit form agent g(? , ? , ?). Furthermore, it is possible to keep open not
only arguments as in these examples but also the target of future calls, as in agent
{STATE}.show_votes(a , b , ?). Then a call will be of the form a.call([s, z]) where s is a
STATE and z is of the type of the last argument of show_votes. This possibility is not
available with C# delegates.

The mechanism does not distinguish between open target and open formal argument.
For example, FUNCTION [INTEGER, TUPLE[INTEGER], INTEGER] declares a func-
tion defined in the class INTEGER, with one open argument INTEGER, and return type
INTEGER. Thus, this declaration can be instantiated with agent i.plus(?), which defines

---

Open arguments are the arguments provided at the invocation of the agent; closed arguments are the
arguments provided at the creation of the agent.
an agent with open formal argument. But this declaration can be also used with
\texttt{agent \{INTEGER\}.plus\(j\)}, which declares an agent with open target.

4.2 Agent Examples and their Verification Challenge

We present some typical applications of agents. To simplify the notation, we assume agents
are procedures and have at most one argument.

4.2.1 Formatter

The first example comes from a paper by Leavens et al. \cite{63} and is recouched in Eiffel
below. It is of particular interest since they describe it as a verification challenge beyond
current techniques. The class \textit{FORMATER} models paragraph formatting with two align-
ment routines. The class \textit{PARAGRAPH} includes a procedure to format the current para-
graph. For illustration purposes, the routines \textit{align_left} and \textit{align_right} require that the
paragraph is not left aligned and not right aligned, respectively. The routines \textit{left_aligned}
and \textit{right_aligned} are pure routines (side effect free) defined in the class \textit{PARAGRAPH},
and return \textit{true} if the paragraph is left aligned or right aligned, respectively. These classes
are implemented in Eiffel as follows:

\begin{verbatim}
class FORMATER
    align_left (p: PARAGRAPH)
        require
            not p. left_aligned
        do
            ... Operations on p ...
        ensure
            p. left_aligned
    end

class PARAGRAPH
    format (proc: PROCEDURE [FORMATTER, PARAGRAPH]; f: FORMATTER)
        do
            proc.call (f, Current)
        end
end
\end{verbatim}
The signature $\text{proc: \textproc{PROCEDURE [FORMATTER, PARAGRAP}}$ \cite{3} declares a procedure $\text{proc}$ with two open arguments (the target of type $\text{FORMATTER}$ and a parameter of type $\text{PARAGRAPH}$). Open arguments are the arguments provided in the invocation of the agent. An example of the use of the $\text{format}$ routine is shown in the routine $\text{apply\_align\_left}$. This routine is implemented as follows:

\begin{verbatim}
apply\_align\_left ( f: \text{FORMATTER}; p: \text{PARAGRAPH})
   \text{require}
      not p. left\_aligned
   \text{do}
      p. format (agent \{ \text{FORMATTER}\}.align\_left , f)
   \text{ensure}
      p. left\_aligned
\end{verbatim}

The verification challenge in this case is to specify and verify the routine $\text{format}$ in an abstract way, abstracting the pre and postcondition of the agent. Then, one should be able to invoke the routine $\text{format}$ with a concrete agent, here $\text{align\_left}$, and to show that the postcondition of $\text{align\_left}$ holds. If the $\text{format}$ routine is called with another routine, say $\text{align\_right}$, one should be able to show that the postcondition of $\text{align\_left}$ holds without modifying the proof of $\text{format}$. Another issue is framing; one should be able to express what the routine $\text{format}$ modifies, but abstracting from the specific routines $\text{align\_left}$ and $\text{align\_right}$. When the routine $\text{format}$ is invoked using the agent $\text{align\_left}$, we should be able to show that $\text{format}$ only modifies what $\text{align\_left}$ modifies.

### 4.2.2 Multi-Level Undo-Redo

The command pattern \cite{40} can be used to implement multi-level undo-redo mechanisms. The standard implementation uses a class $\text{COMMAND}$ with features $\text{execute}$ and $\text{cancel}$. This example involves a history list, of type $\text{LIST [COMMAND]}$, such that it is possible to undo all previously recorded commands through the following routine:

\begin{verbatim}
undo\_all ( history\_list : \text{LIST [COMMAND]})
do
   history\_list . do\_if(agent \{ \text{COMMAND}\}.cancel, agent \{ \text{COMMAND}\}.cancelable)
end
\end{verbatim}

The routine $\text{cancelable}$ returns true if the command can be canceled; in other words it satisfies the precondition of $\text{cancel}$. The iteration routine $\text{do\_if}$ is available to all list

\footnote{To simplify the notation, we do not use $\text{TUPLE}$ in the declaration of the agent. Thus, we write $\text{PROCEDURE [FORMATTER, PARAGRAPH]}$ instead of $\text{PROCEDURE [FORMATTER, TUPLE [FORMATTER, PARAGRAPH]]}$.}
classes through their declaration in the ancestor class *LINEAR*, where its implementation is:

```plaintext
do if (f: PROCEDURE [ANY]; test: PREDICATE [ANY])
do
  from
  start
  until
  after
loop
  if test.item(item) then
    f.call(item)
end
forth
end
```

The declaration `f: PROCEDURE [ANY]` indicates that `f` can come from any class and takes no arguments besides the target; similarly for `test`. (ANY is the most general class, from which all classes descend; cf. “Object” in Java.) The loop follows a standard scheme moving a cursor: `start` brings the cursor to the first element, `forth` moves it by one position, `after` indicates whether the cursor is past the last element, and `item` gives the element at cursor position. The invocation `test.item(item)` calls the agent `test` with the element at the cursor position, and returns the result.

The verification challenge in this example is to reason about the pre- and postcondition of the agent applied to several objects (in this example the elements of the list), where each agent invocation changes the properties of a single object. Verifying these kinds of examples is challenging because the invocation of the agent on one target might change the properties of the other targets. The use of multiple targets also illustrates one of the differences between agents and delegates in C#: applying an agent to multiple objects requires the ability to pass function objects with open target.

### 4.2.3 Archive Example

In this section we describe the *archive* example presented by Leavens et al. [63] and proved by Müller and Ruskiewicz [83]. This example illustrates the application of agents with closed arguments (closed arguments are the arguments of an agent provided at declaration of the agent).

---

4 In `test.item(item)`, the first occurrence of `item` is the routine `item` in class *FUNCTION* (which calls the agent and returns the result); the second occurrence of `item` is the attribute `item` of the class *LIST* (which gives the element at cursor position).
Figure 4.2 presents the implementation of the archive example in Eiffel. The class TAPE_ARCHIVE defines a tape with a routine store which stores objects if the device is loaded. An application of agents passed as parameter is implemented in the class CLIENT, which calls the agent log_file with the string s. Finally, the class MAIN shows an example of the invocation of the routine log in the CLIENT class.

The invocation log_file.call(s) invokes the procedure log_file with the parameter s. The declaration \textit{PROCEDURE[ANY;TAPE]} indicates that log_file is a procedure with closed target of type TAPE and AN open argument of type ANY. The target of the invocation is defined in the creation of the agent. In this example, the target object is \texttt{t} defined in \texttt{agent t.store}.

The verification challenge in this case is to verify the routine log in an abstract way, and being able to show that the precondition of the agent store holds before its invocation. In the routine log, the methodology has to assume that the target is closed but the exact target is unknown.

4.3 Verification Methodology

A verification technique should address both the specification of routines that uses function objects and the verification of invocation of function objects. Section 4.3.1 considers the first issue; the remainder of this section examines the second one. To simplify the methodology, we do not handle exceptions. Extending the methodology to exception handling is part of future work.

4.3.1 Specifying Function Objects

The difficulty of specifying the correctness of agents is that while a variable of an agent type represents a routine, it is impossible to know statically which routine that is. The purpose of agents is to abstract from individual routines. The specification must reflect this abstraction.

What characterizes the correctness of a routine is its precondition and its postcondition. For an agent, these are known abstractly through the functions \textit{precondition} and \textit{postcondition} of class ROUTINE and its descendants. These functions enable us to perform the necessary abstraction on agent variables and expressions. The approach makes it possible for example to equip the routine \textit{format} with a contract:

\footnote{This is a simplification of the declaration in Eiffel. The declaration in Eiffel is \textit{PROCEDURE[TAPE, TUPLE[ANY]]}.}
class TAPE
    save(o: ANY)
    do
    ...  
    end
    -- other routines omitted
end

class TAPE_ARCHIVE
    tape: TAPE
    is_loaded: BOOLEAN
    ensure
        Result = (tape /= void)
    make
        do
            create tape
        end
    store (o: ANY)
    require
        is_loaded
    do
        tape.save (o)
    end
    -- other routines omitted
end

class CLIENT
    log ( log_file : PROCEDURE [ANY; TAPE];
          s: STRING )
    do
        log_file . call (s)
    end
end

class MAIN
    main (c: CLIENT)
    local
        t: TAPE_ARCHIVE
    do
        create t.make
        c.log (agent t.store, "Hello World")
    end
end

Fig. 4.2: Source Code of the Archive Example.
4.3 Verification Methodology

format (proc: PROCEDURE [FORMATTER, PARAGRAPH ]; f: FORMATTER)
  require
    proc.precondition (f, Current)
  do
    proc.call (f, Current)
  ensure
    proc.postcondition (f, Current)
end

Note that the precondition of format uses the routine precondition to query the precondition of the procedure proc.

Finally, we need to specify the routine call in the class ROUTINE. Its specification is the following:

call (target: ANY; p: ANY)
  require
    Current.precondition (target, p)
  ensure
    Current.postcondition (target, p)

4.3.2 Axiomatic Semantics

This section describes an axiomatic semantics for agents. To simplify the semantics, we assume agents with open arguments. In Section 4.4, we show the development of an automatic prover for reasoning about agents. The prover handles open and closed arguments.

Agent Pre- and Postconditions

The semantics uses two functions to model the pre- and postcondition of the agent. The function \( @\text{precondition} \) takes three values (the agent, the target and the parameter) and an object store, and yields the evaluation of the agent’s precondition. The function \( @\text{postcondition} \) takes a second object store to evaluate old expressions. The signatures of these functions are defined as follows:

\[
@\text{precondition} : \text{Value} \times \text{Value} \times \text{Value} \times \text{ObjectStore} \rightarrow \text{Bool} \\
@\text{postcondition} : \text{Value} \times \text{Value} \times \text{Value} \times \text{ObjectStore} \times \text{ObjectStore} \rightarrow \text{Bool}
\]

In the source language, every routine \( pr \) has a pre- and postcondition. The mathematical functions \( @\text{pre}_{pr} \), and \( @\text{post}_{pr} \) evaluates the pre- and postcondition of the routine \( pr \). The signatures of these functions are as follows:

\[\text{We use the prefix } @ \text{ in the mathematical functions to distinguish them from the Eiffel routines.}\]

\[\text{\( \rightarrow \)} \text{ denotes partial functions.}\]
We treat an agent $a$ as a normal variable, which denotes a routine. The following two axioms relates the agent pre- and postcondition with the routine pre- and postcondition, respectively.

**Axiom 12 (Preconditions).**
$\forall t, p : \text{ObjectId}; pr : \text{Routine}; h : \text{ObjectStore} : @\text{precondition}(pr, t, p, h) = @pre_{pr}(t, p, h)$

**Axiom 13 (Postconditions).**
$\forall t, p : \text{ObjectId}; pr : \text{Routine}; h, h' : \text{ObjectStore} : @\text{postcondition}(pr, t, p, h, h') = @pre_{pr}(t, p, h, h')$

In the following, we present an axiomatic semantics for the routine call and for agent initialization.

**Routine call**

To prove calls to agents, represented as calls to the routine $\text{call}$ of the class $\text{ROUTINE}$, it suffices to rely on the specification of this routine. We assume that the implementation of $\text{call}$ in $\text{ROUTINE}$ is correct with respect to this specification. The following axiom defines this assumption:

\[
\{ @\text{precondition}(\text{Current}, t, p, \$) \} \text{ROUTINE:call}(t, p) \{ @\text{postcondition}(\text{Current}, t, p, \$, \$') \}
\]

In the pre- and postcondition of the rule, $\text{Current}$ denotes an agent (the current routine). Furthermore, in the expression $@\text{postcondition}(\text{Current}, t, p, \$, \$')$, $\$'$ denotes the old object store.

This rule makes possible to rely on standard proof techniques, to prove properties of the form

\[
\{ @\text{precondition}(a, t, p, \$) \} \ a.\text{call}(t, p) \ { @\text{postcondition}(a, t, p, \$, \$') \}
\]

So to prove properties of the form $\{ P \} \ a.\text{call}(t, p) \ { Q \}$, it will suffice to prove that $P$ implies $@\text{precondition}(a, t, p, \$)$ and that $@\text{postcondition}(a, t, p, \$, \$')$ implies $Q$.

**Agent Initialization**

The agent initialization is treated as a normal assignment, where the routine $pr$ is assigned to the agent $a$. 
4.4 Automatic Proofs

One of the goals of our verification effort is to support automatic verification of programs using function objects. To fulfill this goal, we have implemented an automatic verifier for agents, called EVE Proofs.

EVE Proofs follows the architecture of the Spec# verifier [8]. Given an Eiffel program, the tool generates a Boogie2 [56] file, and uses the Boogie verifier to prove the generated program. The tool handles a subset of Eiffel with single inheritance, and basic instructions such as assignments, compound, if then else, loops, and routine calls (the tool does not support neither exception handling nor once routines). The tool is integrated to EVE [38] (the Eiffel Verification Environment), and it can be download at http://eve.origo.ethz.ch/. Once the user has specified pre- and post-conditions, and invariants, the verification is completely automatic. Using EVE Proofs we have automatically proved a significant number of examples including the examples presented in Section 4.2.1 and Section 4.2.3.

In the following, we describe the foundations of our verification tool presenting the encoding of our verification methodology into BoogiePL. First, we handle agents with open arguments (Section 4.4.1). Second, we extend the methodology to closed arguments (Section 4.4.2). We describe framing for open and closed arguments (Section 4.4.3 and Section 4.4.4 respectively). Finally, Section 4.4.5 applies the method to the examples from Section 4.2, and section 4.4.6 extends the experiments to more agent-intensive programs which implement graphical user interfaces. More details about the implementation of EVE Proofs see [122].

4.4.1 Reasoning about agents with open arguments

Agent Pre- and Postconditions

The methodology uses two functions to model the pre- and postcondition of the agent. The function $\text{precondition}$ takes three values (the agent, the target and the parameter), and the current heap, and yields the evaluation of the agent’s precondition. The function $\text{postcondition}$ takes a second heap to evaluate old expressions. The signatures of these functions are defined as follows:

$$\text{precondition} : \text{Value} \times \text{Value} \times \text{Value} \times \text{Heap} \rightarrow \text{Bool}$$
$$\text{postcondition} : \text{Value} \times \text{Value} \times \text{Value} \times \text{Heap} \times \text{Heap} \rightarrow \text{Bool}$$
Initializing Agents

Given the agent initialization $a := \texttt{agent } pr$ where $pr$ is a procedure, the methodology generates the following assumptions:

- \texttt{assume } $\forall t,p : \texttt{ObjectId}; h_1 : \texttt{Heap} : \texttt{precondition}(a, t, p, h_1) = \texttt{pre}_{pr}(t, p, h_1)$
- \texttt{assume } $\forall t,p : \texttt{ObjectId}; h_1, h_2 : \texttt{Heap} : \texttt{postcondition}(a, t, p, h_1, h_2) = \texttt{post}_{pr}(t, p, h_1, h_2)$

where $\texttt{pre}_{pr}$ and $\texttt{post}_{pr}$ denotes the pre- and postcondition of the procedure $pr$, $t$ the target object, and $p$ the argument respectively.

Invoking Agents

Invoking an agent $a$ with target $t$ and argument $p$, $a.call(t, p)$, first asserts the precondition of the agent, and then assumes its postcondition. The proof obligations are the followings:

- \texttt{assert } $\texttt{precondition}(a, t, p, \texttt{Heap})$
- $h_0 := \texttt{Heap}$
- \texttt{havoc } $\texttt{Heap}$
- \texttt{assume } $\texttt{postcondition}(a, t, p, \texttt{Heap}, h_0)$

The current heap is denoted by $\texttt{Heap}$. The assignment $h_0 := \texttt{Heap}$ saves the current heap, then $h_0$ is used to evaluate the postcondition of the agent. The \texttt{havoc} command assigns an arbitrary value to the heap.

This translation is based on the translation of pure routines [33, 67]. The novel concepts are the introduction of the functions $\texttt{precondition}$ and $\texttt{postcondition}$ to model the agent pre- and postcondition, and the generation of assumptions for the initialization of the agent, which relates the pre- and postcondition of the agent with the concrete pre- and postcondition of the procedure.

Noninterference

Agents can be declared with open arguments. If the target is open, the same agent can be invoked with different target objects. To reason about these invocations, we need the notion of noninterference. Given a partial function $f : \texttt{Heap} \rightarrow X$, we call footprints the elements of its domain. In this work we only consider functions $f$ such that, for each heap $h$ on which $f$ is defined, there exists a (unique) minimal sub-heap $h_0$, so that the value of $f$ on larger heaps is completely determined from its value on $h_0$. Functions of type $\texttt{Heap} \rightarrow X$ are obtained from preconditions and bodies of agents by fixing the target and

---

*The syntax \texttt{agent } pr is an abbreviation for keeping all arguments open, as in \texttt{agent } pr(?)*
parameter, and from postconditions by also fixing the old heap (we are interested in the footprint expressed in terms of the new heap only).

Given functions \( f : \text{Heap} \rightarrow X \) and \( g : \text{Heap} \rightarrow Y \), we write the noninterference predicate \( f \# g : \text{Heap} \rightarrow \text{Bool} \) which returns true iff both \( f \) and \( g \) are defined and their minimal footprints are disjoint.

We now lift the disjointness predicate \( \# \) to objects. Let \( C \) be a class and \( F \) the set of state functions (preconditions, postconditions with fixed pre-heap, and bodies of features) which it provides. For \( o, o' \in \text{ObjectId} \) objects of class \( C \), we define

\[
o \# o' : \text{Heap} \rightarrow \text{Bool}
\]

\[\triangleq \forall v, v' \in \text{Value}, \forall f, f' \in F. \ (f(o, v) \# f'(o', v'))(h)\]

The role of the \( \# \) predicate is to generalize from concrete mechanisms for establishing noninterference, namely ownership [28, 65, 82], separation logic [111, 97], regional logic [4]. The idea is that each such formalism is sufficiently expressive to imply instances of \( o \# o' \) facts on a per-example basis. We use this \( \# \) predicate in both the specification language and the logic.

### 4.4.2 Reasoning about Closed Arguments

The above section presents a methodology to reason about agents with open arguments. In this section, we extend the methodology to agents with closed arguments. Framing for agents with closed arguments is omitted here, however, it is presented in Section 4.4.4.

To model closed arguments, we introduce two functions:

\[
\$\text{precondition}_1 \text{ and } \$\text{postcondition}_1
\]

These functions yield the evaluation of pre- and postcondition of an agent with one closed argument (either closed target or closed parameter). The function \( \$\text{precondition}_1 \) takes two values (the agent and the open argument) and the current heap, and yields the evaluation of the precondition of the agent. The function \( \$\text{postcondition}_1 \) takes also a second heap to evaluate old expressions. The signature of these functions are defined as follows:

\[
\$\text{precondition}_1 : \text{Value} \times \text{Value} \times \text{Heap} \rightarrow \text{Bool}
\]

\[
\$\text{postcondition}_1 : \text{Value} \times \text{Value} \times \text{Heap} \times \text{Heap} \rightarrow \text{Bool}
\]

To handle arbitrary number of arguments in a routine, say \( n \) arguments, the methodology can be extended by adding the functions \( \$\text{precondition}_0 \ldots \$\text{precondition}_n \) and the functions \( \$\text{postcondition}_0 \ldots \$\text{postcondition}_n \). The functions

\[
\$\text{precondition}_i : \text{Value}^{i+1} \times \text{Heap} \rightarrow \text{Bool}
\]

\[
\$\text{postcondition}_i : \text{Value}^{i+1} \times \text{Heap} \times \text{Heap} \rightarrow \text{Bool}
\]

can be used to model agents with \( i \) open arguments.

\[\text{As a reminder, we assume that routines have only one parameter, although, the methodology can be easily extended.}\]
Initializing Agents

To handle closed arguments, the methodology generates new assumptions using the functions $\text{precondition}_1$ and $\text{postcondition}_1$. In the following, we present these assumptions for closed target and closed arguments.

Closed Target. Given the agent initialization $a := \text{agent } t_1.pr$ where $t_1$ is the closed target, and $pr$ a procedure, the methodology generates the following assumptions:

\begin{align*}
\text{assume } & \forall p : \text{ObjectId}; h_1 : \text{Heap} : \text{precondition}_1(a, p, h_1) = \text{pre}_pr(t_1, p, h_1) \\
\text{assume } & \forall p : \text{ObjectId}; h_1, h_2 : \text{Heap} : \text{postcondition}_1(a, p, h_1, h_2) = \text{post}_pr(t_1, p, h_1, h_2)
\end{align*}

These assumptions quantify only over one parameter, $p$. The target object $t_1$ is known, and it is used in the function $\text{pre}_pr$. The difference with the assumptions generated for open arguments (Section 4.4.1) is that the assumptions for open arguments quantify over both the target and the parameter.

Closed Parameter. Given the agent initialization $a := \text{agent } pr(p_1)$ where $p_1$ is the closed parameter, and $pr$ a procedure, the methodology generates the following assumptions:

\begin{align*}
\text{assume } & \forall t : \text{ObjectId}; h_1 : \text{Heap} : \text{precondition}_1(a, t, h_1) = \text{pre}_pr(t, p_1, h_1) \\
\text{assume } & \forall t : \text{ObjectId}; h_1, h_2 : \text{Heap} : \text{postcondition}_1(a, t, h_1, h_2) = \text{post}_pr(t, p_1, h_1, h_2)
\end{align*}

Invoking Agents

The invocation of an agent with closed arguments takes as arguments the agent and the open parameter. Given the agent $a$ which declares a procedure with one open argument (it can be open target or open parameter) the agent invocation $a.call(p)$ with argument $p$, defines the following proof obligations:

\begin{verbatim}
assert $\text{precondition}_1(a, p, \text{Heap})$
$h_0 := \text{Heap}$
\text{havoc } \text{Heap}$
\text{assume } $\text{postcondition}_1(a, p, \text{Heap}, h_0)$
\end{verbatim}

where $\text{Heap}$ denotes the current heap.

Note that Eiffel does not distinguish between an agent with open target and an agent with open parameter. Both agents are declared with the same notation. Thus, the methodology uses the functions $\text{precondition}_1$ and $\text{postcondition}_1$ to express the precondition and postcondition with open arguments, and then it uses the assumptions generated in the initialization of the agent. An example of the application of open arguments is presented in Section 4.4.5.
4.4 Automatic Proofs

4.4.3 Framing

One of the most interesting parts of routines’ specification is the modifies clause, which defines the locations that are modified by the routine. The problem of defining these locations is known as frame problem. The frame problem has been solved for example using dynamic frames [59, 119]. However, this problem has to be solved for routines that take other routines as arguments (agents). This section presents a solution for framing agents. First, framing is solved for agents with open arguments, and then the methodology is extended for agents with closed arguments.

Framing for Agents with Open Arguments

In Section 4.2.1 we have specified the routine format, however, one needs to define what locations this routine modifies:

format (proc: PROCEDURE [FORMATTER, PARAGRAPH]; f: FORMATTER)
    do
        proc. call (f, Current)
    end

A candidate solution to this problem is to assume that format modifies the target of the agent proc. However, this assumption is too strong since format may only modify a few attributes of proc’s target. Note that format can be invoked with any routine, and each routine might modify different locations.

To solve the frame problem for agents, we adapt dynamic frames. Instead of using a set of locations as in Kassios’s work [59], we define a routine modifies (in the source language) which takes an agent a, its target and argument’s values, and returns the locations modified by the agent a with target t and argument p. Thus, the modifies clause of format can be defined as follows (pre and postconditions are omitted):

format (proc: PROCEDURE [FORMATTER, PARAGRAPH]; f: FORMATTER)
    modify
        modifies (proc, f, Current)
    do
        proc. call (f, Current)
    end

This modifies clause expresses that the routine format modifies the locations that are modified by the procedure proc. Depending of the routine used to invoke format, the function modifies will yield a different set of locations.

Modifies Clauses. We have extended Eiffel with modifies clauses. Each routine contains a modifies clause which is defined as a comma separated list of locations. To express
what locations are modified by an agent, we introduce the function $\text{modifies}$. The definition of modifies clauses and routines declarations is the following:

\[
\text{Modifies}_\text{clause} ::=: \text{Modifies}_\text{clause}, \text{Modifies}_\text{clause} \\
| \text{VarId} \\
| \text{modifies}(\text{VarId, VarId, VarId})
\]

\[
\text{Routine} ::=: \text{RoutineId} (\text{VarId} : \text{Type}) : \text{Type} \\
\quad \text{require} \ \text{BoolExp} \\
\quad \text{modify} \ \text{Modifies}_\text{clause} \\
\quad \text{do} \\
\quad \quad \text{Instr} \\
\quad \text{ensure} \ \text{BoolExp} \\
\quad \text{end}
\]

where \text{boolExp} are boolean expressions, \text{RoutineId} routine identifiers, \text{VarId} variable identifiers, and \text{Instr} instructions.

**Encoding of Modifies Clauses.** To encode the modifies clauses, we introduce a function $\text{modifies}$ which takes an agent $a$, its target and argument’s values, the current heap, an object value $o$, and a field name $f$, and yields true if the agent $a$ with its target and argument modifies the field $f$ of the object $o$. The signature of this function is the following:

\[
\text{modifies} : \text{Value} \times \text{Value} \times \text{Value} \times \text{Heap} \times \text{Value} \times \text{FieldId} \rightarrow \text{Bool}
\]

For example, the modifies clause of the routine \textit{format} can be encoded using this function as follows:

\[
\text{ensures} \ \forall o : \text{ObjectId}; \ fId : \text{FieldId} : \\
\quad \text{not } \text{modifies}(\text{proc, f, Current, Heap, o, fId}) \Rightarrow \text{Heap}[o, fId] = \text{old}(\text{Heap})[o, fId]
\]

This property expresses that for all objects $o$, and all fields $fId$ that are not modified by the agent $\text{proc}$ with the target $f$ and argument $\text{Current}$, the value of the field $o.fId$ in the current heap is equal to the value of $o.fId$ in the old heap. The expression $\text{Heap}[o, fId]$ yields the value of the field $fId$ of the object $o$ in the current heap, and $\text{Heap}$ denotes the current heap.

Generalizing, modifies clauses are list of application of the function $\text{modifies}$ and variable identifiers. Given the modifies clause:

\[
\text{modifies}(a_1, t_1, p_1), ..., \text{modifies}(a_n, t_n, p_n), v_1, ..., v_m
\]

this clause is encoded as:
ensures \( \forall o : \text{ObjectId}; \ fId : \text{FieldId} : \)
\[
\left( \text{not} \$\text{modifies}(a_1, t_1, p_1, \text{Heap}, o, fId) \land \ldots \land \text{not} \$\text{modifies}(a_n, t_n, p_n, \text{Heap}, o, fId) \land o! = v_1 \land o! = v_m \right) \Rightarrow \text{Heap}[o, fId] = \text{old}(\text{Heap})[o, fId]
\]

**Initializing Agents.** To solve the frame problem for agents, we need to link the function \$\text{modifies}(\text{proc}, t, p)\) with the locations that the routine \text{proc}\) modifies. We solve this by applying the same approach to reasoning about agent’s pre- and postcondition. Thus, our methodology generates assumptions of the function \$\text{modifies}\), when the agent is initialized. Given a procedure \(\text{pr}\), the agent initialization \(a := \text{agent} \ \text{pr}\) generates the following assumptions:

\[
\text{assume} \ \forall t, p : \text{ObjectId}; h_1 : \text{Heap} : \$\text{precondition}(a, t, p, h_1) = \$\text{pre}_{\text{pr}}(t, p, h_1)
\]
\[
\text{assume} \ \forall t, p : \text{ObjectId}; h_1, h_2 : \text{Heap} : \$\text{postcondition}(a, t, p, h_1, h_2) = \$\text{post}_{\text{pr}}(t, p, h_1, h_2)
\]
\[
\text{assume} \ \forall t, p, o : \text{ObjectId}; fId : \text{FieldId}; h_1 : \text{Heap} : \$\text{modifies}(a, t, p, h_1, o, fId) = \$\text{modifies}_{\text{pr}}(t, p, o, fId)
\]

The assumptions for the functions \$\text{precondition}\) and \$\text{postcondition}\) are the same as assumptions described in Section 4.4.1. The third assumption relates the function \$\text{modifies}\) with the modifies clause of \(\text{pr}\). The function \$\text{modifies}_{\text{pr}}\) yields \text{true}\) if the procedure \(\text{pr}\) modifies the field \(o.fId\) for the target \(t\) and argument \(p\). For example, assuming that the routine \text{align_left}\) in the class \text{FORMATER}\ (Section 4.2.1) modifies its argument \(p\), then \$\text{modifies}_{\text{align_left}}\) is defined as follows:

\[
$\text{modifies}_{\text{align_left}}(\text{Current}, p, o, fId) \triangleq (o = p)
\]

Generalizing, the function \$\text{modifies}_{\text{pr}}\) takes a target object, an argument, and the object id and field id. The definition of this function can be generated from the modifies clause of each procedure \(\text{pr}\).

### 4.4.4 Framing for Agents with Closed Arguments

**Modifies Clauses.** To define the locations that an agent with closed arguments modifies, we introduce a function \$\text{modifies}_1\). This function takes an agent \(a\) and its open
argument, and returns the locations modified by the agent \( a \) with the argument \( p \). The definition of the modifies clause is extended as follows:

\[
\text{Modifies\_clause ::= } \text{Modifies\_clause, Modifies\_clause} \\
| \text{VarId} \\
| \text{modifies(VarId, VarId, VarId)} \\
| \text{modifies}_1(\text{VarId, VarId})
\]

Using the function \( \text{modifies}_1 \), the modifies clause of the routine \( \text{log} \) (Section [4.2.3]) can be defined as follows:

\[
\text{log } (\logfile : \text{PROCEDURE [ANY;TAPE]; } s : \text{STRING}) \\
\quad \text{modify } \\
\quad \quad \text{modifies}_1(\logfile, s) \\
\quad \text{do} \\
\quad \quad \logfile . \text{call}(s) \\
\quad \text{end}
\]

**Encoding of Modifies Clauses.** The encoding of the modifies clauses follows the same ideas of the above section. We define a function \( \text{$modifies_1$} \) which takes an agent \( a \), its open argument’s value, the current heap, an object value \( o \), and a field name \( fId \), and yields \text{true} if the agent \( a \) with argument \( p \) modifies the field \( o.fId \). The signature of this function is the following:

\[
\text{$modifies_1$} : \text{Value} \times \text{Value} \times \text{Heap} \times \text{Value} \times \text{FieldId} \rightarrow \text{Bool}
\]

For example, the modifies clause of \( \text{log} \) can be encoded as follows:

\[
\text{free ensures } \forall o : \text{ObjectId}; \ fId : \text{FieldId} : \\
\quad \text{not } \text{$modifies_1$(logfile, s, Heap, o, fId)} \Rightarrow \text{Heap}[o, fId] = \text{old}(\text{Heap})[o, fId]
\]

**Initializing Agents.** To conclude with the framing for closed arguments, the methodology generates assumptions in a similar way to the above section. We describe the assumptions generated for initialization of agents with closed target, closed parameter is analogous.

Given the agent initialization \( a := \text{agent } t_1.pr \) where \( t_1 \) is the closed target, and \( pr \) a procedure, the methodology generates the following assumptions:
4.4 Automatic Proofs

assume $\forall p : ObjectId; h_1 : H_{eap} : \text{precondition}_1(a, p, h_1) = \text{pre}_pr(t_1, p, h_1)$
assume $\forall p : ObjectId; h_1, h_2 : H_{eap} : \text{postcondition}_1(a, p, h_1, h_2) = \text{post}_pr(t_1, p, h_1, h_2)$

assume $\forall o, p : ObjectId; fId : FieldId :$ $\text{modifies}_1(a, p, o, fId) = \text{modifies}_pr(t_1, p, o, fId)$

4.4.5 Applications

In this section we study the applicability of our methodology to a range of examples which illustrate challenging aspects of reasoning about function objects.

Formatter Example

In this section, we show how to verify the formatter example presented in Section 4.2.1.

To verify this routine, the methodology generates the following proof obligations:

format$(proc : PROCEDURE[FORMATTER, PARAGRAPH]; f : FORMATTER)$
1 assume $\text{precondition}(proc, f, Current, Heap)$
2 assert $\text{precondition}(proc, f, Current, Heap)$
3 $h_0 := Heap$
4 havoc Heap
5 assume $\text{postcondition}(proc, f, Current, Heap, h_0)$
6 assume $\forall o : ObjectId; fId : FieldId :$
   \text{not} $\text{modifies}(proc, f, Current, Heap, o, fId) \Rightarrow Heap[o, fId] = h_0[o, fId]$
7 assert $\text{postcondition}(proc, f, Current, Heap, h_0)$
8 assert $\forall o : ObjectId; fId : FieldId :$
   \text{not} $\text{modifies}(proc, f, Current, Heap, o, fId) \Rightarrow Heap[o, fId] = h_0[o, fId]$

The pre- and postcondition of format are translated in the lines 1 and 7, respectively. The modifies clause of format is translated in line 8. The agent invocation is translated in the lines 2-6. This translation assumes the postcondition and the modifies clause of call in lines 5 and 6. The proof is straightforward since the assume and assert instructions in lines 1-2, lines 5-7, and lines 6-8 refer to the same heap.

The most interesting case in the verification of function object is the verification of clients that use function objects, such as apply_align_left. Applying our methodology to this routine generates the following assumptions and assertions:
apply_align_left(f : FORMATTER; p : PARAGRAPH)
1 assume not p.$left_aligned
2 a := agent{FORMATTER}.align_left
3 assume \( \forall t_1, p_1 : \text{ObjectId}; h : \text{Heap} : \)
4 \( \$precondition(a, t_1, p_1, h) = \$prealign_left(t_1, p_1, h) \)
5 assume \( \forall t_1, p_1 : \text{ObjectId}; h, h' : \text{Heap} : \)
6 \( \$postcondition(a, t_1, p_1, h, h') = \$postalign_left(t_1, p_1, h, h') \)
7 assume \( \forall o : \text{ObjectId}; fId : \text{FieldId} : \)
8 havoc Heap
9 assume \( \$postcondition(a, f, p, \text{Heap}, h_0) \)
10 assume \( \forall o : \text{ObjectId}; fId : \text{FieldId} : \)
11 assert \( p.$left_aligned \)
12 assert \( \forall o : \text{ObjectId}; fId : \text{FieldId} : o! = p \Rightarrow \text{Heap}[o, fId] = h_0[o, fId] \)

Similar to the previous example, lines 1 and 11 are generated by the translation of the pre- and postcondition; line 12 is the translation of the modifies clause. The declaration agent \{FORMATTER\}.align_left generates lines 2-5. The precondition and postcondition of the routine align_left is denoted by \$prealign_left and \$postalign_left respectively; the modifies clause of align_left is denoted by \$modifyalign_left. The invocation of the routine \text{format} produces lines 6-10. The current heap is stored in \( h_0 \) in line 7 to be able to evaluate the postcondition in line 9.

The key points in the proof are the assert instructions at lines 6, 11 and 12. By the definition of \$prealign_left, \$postalign_left, and \$modifyalign_left we know:

\[ \forall t_1, p_1 : \text{ObjectId}; h : \text{Heap} : \$prealign_left(t_1, p_1, h) = \text{not} p_1.$left_aligned \]  \hfill (4.1)
\[ \forall t_1, p_1 : \text{ObjectId}; h, h' : \text{Heap} : \$postalign_left(t_1, p_1, h, h') = p_1.$left_aligned \]  \hfill (4.2)
\[ \$modifyalign_left(\text{Current}, p, o, fId) = o! = p \]  \hfill (4.3)

In particular, \( \$prealign_left(f, p, \text{Heap}) = \text{not} p.$left_aligned \). Then, the assertion at line 6 is proven using the assumptions at lines 1 and 3, and (4.1). The assertion at line 11 is proven in a similar way using the assumptions at lines 4 and 9, and (4.2). Finally, the assertion at line 12 is proven in a similar way using the assumptions at lines 5 and 10, and (4.3).
4.4 Automatic Proofs

Multi-Level Undo-Redo Example

In this section, we discuss how to prove the routine do_if presented in Section 4.2.2. The proof of routine undo_all is similar to apply_align_left.

The first step is to give an specification of the routine do_if. The idea of do_if(f, test) is to execute f on all the elements which satisfy test. There is, however, a problem in specification of this routine in the EiffelBase library: test can be arbitrary and it might not imply that the precondition of f holds. A first try at improving the contract of the routine do_if could be the following:

\[
\text{do_if (f: \text{PROCEDURE [ANY]; test: \text{FUNCTION [ANY]}})}
\]

\[
\begin{align*}
\text{require} & \quad \forall 1 \leq i \leq \text{count}: \text{test}(\text{ith}(i)) \implies f.\text{precondition}(\text{ith}(i)) \\
\text{ensure} & \quad \forall 1 \leq i \leq \text{count}: (\text{old test}(\text{ith}(i)) \implies f.\text{postcondition}(\text{ith}(i)))
\end{align*}
\]

This contract uses two more features of the class LINEAR: ith and count. Function ith returns the i-th element of the current structure, and attribute count contains the length of the structure.

However, this improved contract is still not sufficient. Consider the list \(l = [c, c]\) where the command c satisfies the query cancelable. Consider an invocation of do_if with the agents cancel and cancelable. Since the precondition of cancel holds for all the elements of the list \(l\), the routine can be invoked. Thus, the first agent is invoked. But assume that this invocation breaks the property cancelable, then since the list contains two repeated elements, the second agent invocation does not satisfy cancelable and an exception is triggered.

The root of the problem lies in the fact that the invocation of the agent on one element of the list could break the precondition of the next element. To prevent the problem we must impose further conditions. We do that using the noninterference predicate \# presented in Section 4.4.1. These assertions are then treated as proof obligations which need to be discharged by appropriate mechanisms in the target language. Example mechanisms are richer type systems based on ownership [25, 28] (as in our experiments with Spec#), or richer program logics based on separation logic [111, 97]. In other words, the \# operator specifies that objects do not interfere (they occupy disjoint memory in case of separation logic, or they belong to different contexts in case of ownership). Here, we use the property that agent invocations only modify the target object, and that noninterference still holds after the invocation. Using this extension, we can have another go at writing the contract for do_if:
do_if \ (f: \text{PROCEDURE \ [ANY]; test: \text{FUNCTION \ [ANY]})}

\text{require}
\quad \text{forall } 1 \leq i \leq \text{count: \ test(i)} \implies f.\text{precondition}(i)
\quad \text{and}
\quad \text{forall } 1 \leq i < j \leq \text{count: \ ith(i) \# ith(j)}

\text{ensure}
\quad \text{forall } 1 \leq i \leq \text{count: \ old \ test(i) \implies f.\text{postcondition}(i)}

The new precondition says in addition that there is no interference between the elements of the list. We now present a sketch of the proof of the routine \text{do_if}. Section 4.4 shows how this example is encoded and automatically proved in Spec#.

Let \text{do_if.pre} be the precondition of the routine \text{do_if} and \text{loop.invariant} be the loop invariant defined as follows:

\[
\text{do_if.pre} \triangleq \text{Inv}(\alpha) \land \text{forall } 1 \leq i < j \leq \text{count : ith(i)#ith(j)} \\
\text{loop.invariant} \triangleq 1 \leq j \leq \text{count + 1} \land \text{forall } j \leq i < k \leq \text{count : ith(i)#ith(k)} \\
\text{forall } 1 \leq i \leq j - 1 : (\beta_i \Rightarrow \text{postcondition}(f, \alpha_i, \text{Heap}, h_0) \\
\text{forall } j \leq i \leq \text{count : (\alpha_i.$test \Rightarrow \text{precondition}(f, \alpha_i, \text{Heap})}
\]

We use the auxiliary variable \(\alpha\) to represent the current structure, which is a sequence of length \text{count}. The \(i\)-th element is denoted \(\alpha_i\). The expression \(\text{Inv}(\alpha)\) denotes the invariant of the class parameterized by the sequence \(\alpha\). To translate away the \text{old} operator, we introduce auxiliary variable \(\beta\), also denoting a sequence of length \text{count}. In the precondition, we assume \(\alpha_i.$test = \beta_i\). In the postcondition, expression \text{old} \(\alpha_i.test\) is translated as \(\beta_i\).

Figure 4.3 presents the sketch of the proof of the routine \text{do_if}. The auxiliary variable \(j\) represents the index of the current structure. After the invocation \(f.call(item)\), we have to show that there is still no interference between the elements of the list. This is exactly the property of the operator \(\#\), which we have introduced in Section 4.4.1. The assertion at line 6 is proved using the loop invariant. The loop invariant is re-established using the property of the \(\#\) operator.

\textbf{Archive Example}

To verify the archive example, we apply the methodology described in Section 4.4.2. The most interesting part is the proof of the routine \text{main}.

The proof for the routine \text{log} is similar to the proof of the \text{format} routine. The only change is the use of the function \text{precondition}\(1\) which takes only three arguments (the procedure \text{log_file}, the string \(s\) and the heap). The proof obligations are:
4.4 Automatic Proofs

\begin{verbatim}
do_if (f : PROCEDURE[ANY]; test : FUNCTION[ANY])
do
assume do_if_pre
h0 := Heap
from start until after loop
if test(item) then
  assert $precondition(f, \alpha_j, Heap)
  h1 := Heap
  f.call(item)
  assume $postcondition(f, \alpha_j, Heap, h1)
end
assert loop invariant
end
assert (Inv(\alpha) \land \forall 1 \leq i < k \leq count : ith(i) \neq ith(k) \land
(forall 1 \leq i \leq count : (\beta_j \Rightarrow $postcondition(f, \alpha_j, Heap, h0)))
end
\end{verbatim}

Fig. 4.3: Sketch of the Proof of the Routine do_if.

\begin{verbatim}
log(log_file : PROCEDURE[ANY; TAPE]; s : STRING)
assume $precondition1(log_file, s, Heap)
assert $precondition1(log_file, s, Heap)
log_file.call(s)
\end{verbatim}

The proof of routine main translates the agent initialization in lines 3-5. The function $precondition1$ is used to express the precondition of the agent with closed target. Using the assumption at line 4 and the knowledge of line 2, we can prove the assert instruction at line 7. The sketch of the proof is:

\begin{verbatim}
main(c : CLIENT)
create t.make
assert t.$is\_loaded
a := agent t.store
assume \forall p_1 : ObjectId; h : Heap :
$precondition1(a, p_1, h) = $prestore(a, t, p_1, h)
assume \forall p_1 : ObjectId; h, h' : Heap :
$postcondition1(a, p_1, h, h') = $poststore(a, t, p_1, h, h')
assert $precondition1(a, "HelloWorld", Heap)
c.log(a, "HelloWorld")
\end{verbatim}
4.4.6 Experiments

Using the implemented tool, *EVE Proofs*, we have automatically proven a suite of examples: the examples presented in Section 4.2, and several more extensive agent-intensive programs to model graphical user interfaces. The examples can be downloaded from [http://se.ethz.ch/people/tschannen/examples.zip](http://se.ethz.ch/people/tschannen/examples.zip). The experiments were run on a machine with a 2.71 GHz dual core Pentium processors with 2GB of RAM.

Table 4.1 presents the results of the experiments. For each example, the table shows the number of classes, routines, agents, and lines of code in Eiffel, as well as the number of lines of the encoding in Boogie. The last column shows the running time of Boogie (the dominant factor in the verification).

The formatter and archiver examples have been discussed in the previous sections. The third example is a typical implementation of the command pattern [40]. It defines a command class that uses an agent to store an action, which will be executed when the command’s execute function is called. This pattern is also used in the calculator and ATM examples, which model applications using graphical user interfaces (GUI). The calculator example implements the GUI of a simple calculator with buttons for the digits, and basic arithmetic operations such as addition, subtraction, and multiplication. The ATM example implements a GUI for an ATM machine, and it also implement client code where a pin number is entered, and money is deposited and withdrawn from an account. These two examples are of particular interest because the GUI libraries in Eiffel typically use agents to react on events.

The cell/recell example is an extension of an example by Parkinson and Bierman [102] with agents. The counter example implements a simple counter class with increase and decrease operations. The last example defines a class hierarchy for integer sequences introducing an arithmetic and Fibonacci sequence.
4.5 Related Work

Jacobs [54] as well as Müller and Ruskiewicz [83] extend the Boogie verification methodology to handle C# delegates. They associate pre- and postconditions with each delegate type. When the delegate type is instantiated, they prove that the specification of the method refines the specification of the delegate type. At the call site, one has to prove the precondition and may assume the postcondition of the delegate. By contrast, the methodology presented here “hides” the specification behind abstract predicates. Callers will in general require the predicates to hold that they need in order to call an agent. The approach taken by Jacobs, Müller, and Ruskiewicz splits proof obligations into two parts, the refinement proof when the delegate is instantiated and the proof of the precondition when the delegate is called. This split makes it difficult to handle closed parameters, in particular, the closed receiver of C# delegates. Both previous works use some form of ownership [65] to ensure that the receiver of a delegate instance has the properties required by the method underlying the delegate. Our methodology requires only one proof obligation when the agent is called and, avoids the complications and restrictions of ownership and can be generalized to several closed parameters more easily.

Börger et al. [25] present an operational semantics of C# including delegates. The semantics is given using abstract state machines. However, this work does not describe how to apply this model to specify and verify C# programs.

Contracts have been integrated into higher-order functions. Findler et al. [39] integrate contracts using a typed lambda calculus with assertions for higher-order functions. Honda et al. [18, 51] introduce a sound compositional program logic for higher-order functions. However, these solutions cannot be applied to object-oriented languages with their use of the heap and side effects.

A key issue of reasoning about object-oriented programs is framing, that is, how to conclude which heap changes affect which predicates. In this chapter, we simply assumed a noninterference predicate # without prescribing a particular way of enforcing it. Suitable candidates are separation logic [111, 97, 26], dynamic frames [59, 119, 118], or regions [4]. Separation logic offers separating conjunction to express noninterference. Both dynamic frames and regions effect specifications for predicates and routines.

Distefano and Parkinson [36] show the applicability of abstract predicates implementing a tool to verify Java programs. The tool handles several design patterns such as the visitor pattern, the factory pattern, and the observer pattern. The predicates we use for the preconditions and postconditions of agents are inspired by abstract predicates. Even though Parkinson and Bierman’s work and Distefano and Parkinson’s work do not handle function objects, we believe that the ideas presented in this chapter also apply to their setting.
Birkedal et al. [21] present higher-order separation logic, a logic for a second-order programming language, and use it to verify an implementation of the Observer pattern [60]. In contrast to separation logic, the methodology presented in this chapter works with standard first-order theorem provers.

Our encoding of the routines \textit{precondition} and \textit{postcondition} is based on previous work on pure routines by Darvas [32], and Leino and Müller [67].
Part III

Proof Transformations
Chapter 5

The CIL Language and its Logic

Our proof-transforming compiler generates proofs in a logic for CIL. This chapter describes a subset of the CIL language and the bytecode logic used in the proof transformation. The subset of the CIL language includes push and pop instructions, branch instructions, fields, object creation, method invocations, and exception handling using *try catch*, and *try finally* instructions. The bytecode logic is based on the logic developed by Bannwart and Müller [6, 5]. The only contribution of this chapter is the extension of their logic to exception handling. This contribution is presented in Section 5.2.3.

5.1 The CIL Bytecode Language

The bytecode language consists of interfaces and classes. Each class consists of methods and fields. Methods are implemented as method bodies consisting of a sequence of labeled bytecode instructions. Bytecode instructions operate on the operand stack, local variables (which also include parameters), and the heap.

The bytecode language we use is a slight variant of CIL. We treat local variables and routine arguments using the same instructions. Instead of using an array of local variables like in CIL, we use the name of the source variable. Furthermore, to simplify the translation, we assume the bytecode language has a type boolean. The bytecode instructions and their informal description are the following (more detail about CIL see [43]).

- **ldc** \( v \): pushes a constant \( v \) onto the stack.
- **ldloc** \( x \): pushes the value of a local variable or a method parameter \( x \) onto the stack.
- **stloc** \( x \): pops the top element off the stack and assigns it to \( x \).
• \( op_f \): assuming that \( f \) is a function that takes \( n \) input values to \( m \) output values, \( op_f \) removes the \( n \) top elements from the stack by applying \( f \) to them, and puts the \( m \) output values onto the stack. We write \( \text{bin}_{op} \) if \( op \) is a binary function. We consider the following binary instructions:

1. \( \text{add}, \text{rem}, \text{mul}, \text{div} \): these instructions take 2 input values, and remove the 2 top elements from the stack applying the operation addition, subtraction, multiplication or division respectively, and put the result onto the stack.

2. \( \text{ceq}, \text{clt}, \text{cgl} \): these instructions take 2 input values, remove the 2 top elements from the stack applying the operation equal, less than, greater than respectively, and put the result onto the stack.

• \( \text{brtrue} \ l \): transfers control to the point \( l \) if the top element of the stack is \textit{true} and unconditionally pops it.

• \( \text{brfalse} \ l \): transfers control to the point \( l \) if the top element of the stack is \textit{false} and unconditionally pops it.

• \( \text{br} \ l \): transfers control to the point \( l \).

• \( \text{leave} \ l \): exits from \texttt{.try catch} and \texttt{.try .finally} blocks transferring control to the point \( l \).

• \( \text{endfinally} \): exits the \texttt{.finally} block transferring control to the end of the block.

• \( \text{checkcast} \ T \): checks whether the top element is of type \( T \) or a subtype thereof.

• \( \text{newobj} \) instance \texttt{void Class::ctor()}. \( T \): allocates a new object of type \( T \) and pushes it onto the stack.

• \( \text{callvirt} \ M \) and \( \text{call} \ M \) : invokes the method \( M \) on an optional object reference and parameters on the stack and replaces these values by the return value of the invoked method (if \( M \) returns a value). The instruction \texttt{call} invokes non-virtual and static methods, \texttt{callvirt} invokes virtual methods. The code depends on the actual type of the object reference (dynamic dispatch).

• \( \text{ldfld} \ F \): replaces the top element by its field \( F \) (an instance field).

• \( \text{ldsfld} \ F \): replaces the top element by its field \( F \) (a static field).

• \( \text{stfld} \ F \): sets the field \( F \) (an instance field) of the object denoted by the second-topmost element to the top element of the stack, and pops both values.

• \( \text{stsfld} \ F \): sets the field \( F \) (a static field) of the object denoted by the second-topmost element to the top element of the stack, and pops both values.
• ret: returns to caller.

• nop: has no effect.

• .try { seq₁ } catch T { seq₂ }: executes the sequence of CIL instructions seq₁. If any instruction in seq₁ throws an exception of type T, control is transferred to the first instruction in seq₂.

• .try { seq₁ } .finally { seq₂ }: executes the sequence of CIL instructions seq₁, and then it executes the sequence seq₂ (even if an exception is triggered in seq₁).

Example

Figure 5.1 shows an example of a bytecode program. This program is the result of the compilation of the function sum presented in Section 3.1.5. The default initialization of the variables i and Result is compiled in lines IL001 – IL004. The from body is compiled in lines IL005 – IL009. The instruction at line IL009 transfers control to the label IL020 where the until expression is evaluated. This expression is compiled in lines IL020 – IL025. At label IL025, if the until expression evaluates to false, control is transferred to label IL012. Lines IL012 – IL019 translate the body of the loop.

5.2 A Bytecode Logic for CIL

The bytecode logic we use is an extension of the logic developed by Bannwart and Müller [6]. It is a Hoare-style program logic, which is similar in its structure to the source logic. In particular, both logics treat methods in the same way, contain the same language-independent rules, and triples have a similar meaning. These similarities make proof transformation feasible.

In the following sections, we present the bytecode logic. Section 5.2.1 and Section 5.2.2 are based on Bannwart and Müller’s logic. Section 5.2.3 is a contribution of this chapter.

5.2.1 Method and Instruction Specifications

Properties of methods are expressed by method specifications of the form \{P\} m \{Qₙ, Qₑ\} where \(Qₙ\) is the postcondition after normal termination, \(Qₑ\) is the exceptional postcondition, and \(m\) is a virtual method \(Tₙ\) or a method implementation \(Tₚₙ\). Properties of method bodies are expressed by Hoare triples of the form \{P\} comp \{Q\}, where P, Q are first-order formulas and comp is a method body. The triple \{P\} comp \{Q\} expresses the following refined partial correctness property: if the execution of comp starts in a state satisfying P, then (1) comp terminates in a state where
.method public int sum (int n) {
    IL001 : ldc 0
    IL002 : stloc Result
    IL003 : ldc 0
    IL004 : stloc i
    // from body
    IL005 : ldc 1
    IL006 : stloc Result
    IL007 : ldc 2
    IL008 : stloc i
    IL009 : br IL020
    // loop body
    IL010 : ldloc Result
    IL011 : ldloc i
    IL012 : add
    IL013 : stloc Result
    IL014 : add
    IL015 : stloc Result
    IL016 : ldloc i
    IL017 : ldc 1
    IL018 : add
    IL019 : stloc i
    // until expression
    IL020 : ldloc i
    IL021 : ldloc n
    IL022 : ldc 1
    IL023 : add
    IL024 : ceq
    IL025 : brfalse IL012
    IL026 : ret Result

Fig. 5.1: Bytecode Program compiled from the Function sum presented on Section 3.1.5.

Q holds, or (2) comp aborts due to errors or actions that are beyond the semantics of the programming language, or (3) comp runs forever.

In Bannwart and Müller’s logic, method specifications have the form \{P\} m \{Q\}. We have extended the logic to handle exceptions; for this reason, method specifications have the form \{P\} m \{Q_n, Q_e\}.

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

The meaning of an instruction specification cannot be defined in isolation. The instruction specification \{\(E_l\}\} l : I_l expresses that if the precondition \(E_l\) holds when the program counter is at position \(l\), then the precondition of \(I_l\)’s successor instruction holds after normal termination of \(I_l\).
5.2 A Bytecode Logic for CIL

5.2.2 Rules

Rules for Instruction Specifications

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

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

where \( wp(I_l) \) denotes the local weakest precondition of instruction \( I_l \). The rule specifies that \( E_l \) (the precondition of \( I_l \)) has to imply the weakest precondition of \( I_l \) with respect to all possible successor instructions of \( I_l \). The precondition \( E_l \) denotes the precondition of the instruction \( I_l \). The precondition \( E_{l+1} \) denotes the precondition of \( I_l \)'s successor instruction. Table 5.1 shows the definition of \( wp \).

<table>
<thead>
<tr>
<th>( I_l )</th>
<th>( wp(I_l) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>ldc ( v )</td>
<td>( \text{unshift}(E_{l+1}[v/s(0)]) )</td>
</tr>
<tr>
<td>ldloc ( x )</td>
<td>( \text{unshift}(E_{l+1}[x/s(0)]) )</td>
</tr>
<tr>
<td>stloc ( x )</td>
<td>( (\text{shift}(E_{l+1}))[s(0)/x] )</td>
</tr>
<tr>
<td>binop</td>
<td>( (\text{shift}(E_{l+1}))[s(1) \ op s(0)/s(1)] )</td>
</tr>
<tr>
<td>brtrue ( l' )</td>
<td>( (\neg s(0) \Rightarrow \text{shift}(E_{l+1})) \land (s(0) \Rightarrow \text{shift}(E_{l'})) )</td>
</tr>
<tr>
<td>brfalse ( l' )</td>
<td>( (s(0) \Rightarrow \text{shift}(E_{l+1})) \land (\neg s(0) \Rightarrow \text{shift}(E_{l'})) )</td>
</tr>
<tr>
<td>br ( l' )</td>
<td>( E_v )</td>
</tr>
<tr>
<td>leave ( l' )</td>
<td>( E_v )</td>
</tr>
<tr>
<td>nop</td>
<td>( E_{l+1} )</td>
</tr>
<tr>
<td>checkcast ( T )</td>
<td>( E_{l+1} \land \tau(s(0)) \preceq T )</td>
</tr>
<tr>
<td>newobj ( T )</td>
<td>( \text{unshift}(E_{l+1}[\text{new}($, T)/s(0), $T]/$] )</td>
</tr>
<tr>
<td>ldfld ( T@a )</td>
<td>( E_{l+1}[$\langle iv(s(0), T@a)\rangle/s(0)] \land s(0) \neq \text{null} )</td>
</tr>
<tr>
<td>stfld ( T@a )</td>
<td>( (\text{shift}^2(E_{l+1}))[$\langle iv(s(1), T@a) := s(0)\rangle/$] \land s(1) \neq \text{null} )</td>
</tr>
<tr>
<td>ret</td>
<td>true</td>
</tr>
</tbody>
</table>

Tab. 5.1: Definition of function \( wp \).

Within an assertion, the current stack is referred to as \( s \), and its elements are denoted by non-negative integers: element 0 is the topmost element, etc. The interpretation \([E_l]: State \times Stack \to Value\) for \( s \) is defined as follows:

\[ [E_l]_s : State \times Stack \to Value \]

\[ [E_l]_s \]
\[ [s(0)] (S, (\sigma, v)) = v \text{ and } [s(i + 1)] (S, (\sigma, v)) = [s(i)](S, \sigma) \]

The functions \( \text{shift} \) and \( \text{unshift} \) define the substitutions that occur when values are pushed onto and popped from the stack, respectively. Their definitions are the following:

\[
\begin{align*}
\text{shift}(E) &= E[s(i + 1)/s(i) \text{ for all } i \in \mathbb{N}] \text{ and } \\
\text{unshift} &= \text{shift}^{-1} \\
\text{shift}^n &= n \text{ consecutive applications of } \text{shift}.
\end{align*}
\]

### Connecting Instruction and Method Specifications

To show that a CIL method \( m \) satisfies its specification, we have to prove that all the instruction specifications of \( m \) holds, and that the precondition of \( m \) implies the precondition of the first CIL instruction, and that the postcondition of the last instruction implies the normal postcondition of \( m \). Instruction and method specifications are connected using the following rule:

\[
\begin{align*}
P &\Rightarrow E_0 & E_{|\text{body}(T@m)|-1} &\Rightarrow Q_n \\
\forall i : \{0, \ldots, \text{ body}(T@m) \} : A &\vdash \{E_i\} : I_i \\
A &\vdash \{P\} & \text{ body}(T@m) &\{Q_n, Q_e\}
\end{align*}
\]

\( \{P\} \text{ body } (T@m) \{Q_n, Q_e\} \) must be an admissible method specification, in particular \( P, Q_n, Q_e \) must not refer to local variables.

### Rules for Method Specifications

The rules for method specifications are identical to Poetzsch-Heffter and Müller’s \[108\] rules in the source logic (presented in Section 3.1.4). In the following, we present the routine implementation rule, the class rule, and the subtype rule.

**Routine Implementation Rule:**

\[
\begin{align*}
A, \{P\} &\vdash \{T@m, \{Q_n, Q_e\}\} &\vdash \{P\} &\text{ body}(T@m) &\{Q_n, Q_e\} \\
A &\vdash \{P\} & T@m &\{Q_n, Q_e\}
\end{align*}
\]
5.2 A Bytecode Logic for CIL

Class Rule:

\[
\begin{align*}
A \vdash & \left\{ \tau(\text{Current}) = T \land P \right\} \impl(T, m) \left\{ Q_n, Q_e \right\} \\
A \vdash & \left\{ \tau(\text{Current}) < T \land P \right\} \quad T:m \quad \left\{ Q_n, Q_e \right\} \\
A \vdash & \left\{ \tau(\text{Current}) \preceq T \land P \right\} \quad T:m \quad \left\{ Q_n, Q_e \right\}
\end{align*}
\]

Subtype Rule:

\[
A \vdash \left\{ P \right\} \quad S \preceq T \\
\quad S:m \quad \left\{ Q_n, Q_e \right\}
\]

Language-Independent Rules

The bytecode logic has the same language-independent rule as the source logic. However, these rules are only applied to virtual routines \( T:m \), or routine implementations \( T@m \).

Figure 5.2 shows the language-independent rules. The false axiom allows us to prove anything assuming \( \text{false} \). In the assumpt-axiom, assuming that a triple \( A \) holds, one can conclude the same triple. The assumpt-intro-axiom allows us to introduce a triple \( A_0 \) in the hypothesis. The assumpt-elim-axiom allows eliminating a triple \( A_0 \) in the hypothesis.

The strength rule allows proving a Hoare triple with an stronger precondition if the precondition \( P' \) implies the precondition \( P \), and the Hoare triple can be proved using the precondition \( P \). The weak rule is similar but it weakens the postcondition. This rule can be used to weaken both the normal postcondition \( Q_n \), and the exceptional postcondition \( Q_e \). The conjunction and disjunction rule, given two proofs for the same instruction but using possible different pre- and postconditions, it concludes the conjunction and disjunction of the pre- and postcondition respectively. The invariant rule conjuncts \( W \) in the precondition and postcondition assuming that \( W \) does not contain neither program variables or $.

The substitution rule substitutes \( Z \) by \( t \) in the precondition and postcondition. Finally, the all-rule and ex-rule introduces universal and existential quantifiers respectively.

5.2.3 Exception Handling

The exception handling model of CIL is .try catch and .try .finally structures. These structures have a similar semantics to the try-catch and try-finally instructions in C# and Java. The body of the .try, catch and .finally blocks are a sequence of CIL instructions. Control can only exit a .try or catch block by triggering an exception or by the special
The CIL Language and its Logic

Assumpt-axiom

\[ \mathcal{A} \vdash A \]

Assumpt-intro-axiom

\[ \mathcal{A} \vdash A \]
\[ \mathcal{A}_0 \vdash A \]

Strength

\[ P' \Rightarrow P \]
\[ \mathcal{A} \vdash \{ P \} \quad s_1 \quad \{ Q_n, Q_e \} \]
\[ \mathcal{A} \vdash \{ P' \} \quad s_1 \quad \{ Q_n, Q_e \} \]

Conjunction

\[ A \vdash \{ P' \} \quad s_1 \quad \{ Q'_n, Q'_e \} \]
\[ A \vdash \{ P'' \} \quad s_1 \quad \{ Q''_n, Q''_e \} \]
\[ A \vdash \{ P' \land P'' \} \quad s_1 \quad \{ Q'_n \land Q''_n, Q'_e \land Q''_e \} \]

Invariant

\[ A \vdash \{ P \} \quad s_1 \quad \{ Q_n, Q_e \} \]
\[ A \vdash \{ P \land W \} \quad s_1 \quad \{ Q_n \land W, Q_e \land W \} \]
where \( W \) is a \( \Sigma \) – formula, i.e. does not contain program variables or $.$

all-rule

\[ A \vdash \{ P[Y/Z] \} \quad s_1 \quad \{ Q_n, Q_e \} \]
\[ A \vdash \{ P[Y/Z] \} \quad s_1 \quad \{ \forall Z : Q_n, \forall Z : Q_e \} \]
where \( Z, Y \) are arbitrary, but distinct logical variables.

False axiom

\[ \vdash \{ \text{false} \} \quad s_1 \quad \{ \text{false}, \text{false} \} \]

Assumpt-elim-axiom

\[ \mathcal{A}_0 \vdash A \]
\[ \mathcal{A}_0, \mathcal{A} \vdash A \]

Weak

\[ A \vdash \{ P \} \quad s_1 \quad \{ Q'_n, Q'_e \} \]
\[ Q_n \Rightarrow Q'_n \]
\[ Q_e \Rightarrow Q'_e \]
\[ A \vdash \{ P \} \quad s_1 \quad \{ Q'_n, Q'_e \} \]

Disjunction

\[ A \vdash \{ P' \} \quad s_1 \quad \{ Q'_n, Q'_e \} \]
\[ A \vdash \{ P'' \} \quad s_1 \quad \{ Q''_n, Q''_e \} \]
\[ A \vdash \{ P' \lor P'' \} \quad s_1 \quad \{ Q'_n \lor Q''_n, Q'_e \lor Q''_e \} \]

Substitution

\[ A \vdash \{ P \} \quad s_1 \quad \{ Q_n, Q_e \} \]
\[ A \vdash \{ P[t/Z] \} \quad s_1 \quad \{ Q_n[t/Z], Q_e[t/Z] \} \]
where \( Z \) is an arbitrary logical variable and \( t \) a \( \Sigma \) – term.

ex-rule

\[ A \vdash \{ P \} \quad s_1 \quad \{ Q_n[Y/Z], Q_e[Y/Z] \} \]
\[ A \vdash \{ \exists Z : P \} \quad s_1 \quad \{ Q_n[Y/Z], Q_e[Y/Z] \} \]
where \( Z, Y \) are arbitrary, but distinct logical variables.

Fig. 5.2: Language-Independent Rules for the Bytecode Logic
5.2 A Bytecode Logic for CIL

instruction leave. Control can only exit a .finally block by triggering an exception or by the special instruction endfinally. It is not possible to branch into .try, catch and .finally blocks.

Since the only way to exit these block is triggering an exception or using the leave and endfinally instructions, we assume that the last instruction of the .try and catch blocks is a leave instruction, and the last instruction of a .finally block is a endfinally instruction. This assumption is made to simplify the presentation of the rules. The syntax of these instructions is as follows:

\[
\begin{align*}
\text{CilInstruction} & ::= \text{Instruction} \quad \text{// CIL instructions as described in Section 5.1} \\
& | \text{TryCatchBlock} \\
& | \text{TryFinallyBlock} \\
\text{TryCatchBlock} & ::= \text{.try} \{ \\
& \text{list_of CilInstruction} \\
& \text{Label : leave Label} \\
& \}\ \\
& | \text{catch CilType} \{ \\
& \text{list_of CilInstruction} \\
& \text{Label : leave Label} \\
& \}\ \\
\text{TryFinallyBlock} & ::= \text{.try} \{ \\
& \text{list_of CilInstruction} \\
& \text{Label : leave Label} \\
& \}\ \\
& | \text{.finally} \{ \\
& \text{list_of CilInstruction} \\
& \text{Label : endfinally} \\
& \}\ 
\end{align*}
\]

Formalization

If an exception is triggered in a .try block, control is transferred to the corresponding catch block. We introduce the function handler, which returns the label where the exception is caught. To formalize this function, we use an exception table similar to the JVM exception table. This table can be constructed from the CIL program. The definition of this table is the following:

\[
\begin{align*}
\text{ExcTable} & ::= \text{List}[\text{ExcTableEntry}] \\
\text{ExcTableEntry} & ::= [\text{start : Label, end : Label, target : Label, excType : Type}] \\
\end{align*}
\]

In the type ExcTableEntry, the first label is the starting label of the exception line, the second denotes the ending label, and the third is the target label. An exception of type $T_1$
thrown at line \( l \) is caught by the exception entry \([l_{\text{start}}, l_{\text{end}}, l_{\text{targ}}, T_2]\) if and only if \( l_{\text{start}} \leq l < l_{\text{end}} \) and \( T_1 \preceq T_2 \). Control is then transferred to \( l_{\text{targ}} \).

Given a CIL program, the definition of the exception table is given by the .try, catch, and .finally blocks. We use the function \textit{table} to build the exception table. Its definition is the following:

\[
\text{table} : \text{CilProgram} \rightarrow \text{ExcTable} \\
\text{table} [\] = [] \\
\text{table} (\text{.try}\{ w \} \text{catch } T \{ z \}) \quad \text{#xs} \quad l \ T = [l_n, l_{m+1}, l_i, T] + \text{table } w + \text{table } z + \text{table } xs \\
\text{table} (\text{.try}\{ w \} \text{.finally } \{ z \}) \quad \text{#xs} \quad l \ T = [l_n, l_{m+1}, l_i, \text{any}] + \text{table } w + \text{table } z + \text{table } xs \\
\text{table } x\#xs = \text{table } xs \\
\]

where \( w = l_n : \text{Inst}_n, ..., l_m : \text{Inst}_m \) and \( z = l_i : \text{Inst}_i, ..., l_j : \text{Inst}_j \)

The function \textit{handler} takes the exception table, the type of the current exception, and the label where the exception is triggered, and it returns the label where control is transferred. The function \textit{handler} is defined as follows:

\[
\text{handler} : \text{ExcTable} \times \text{Label} \times \text{Type} \rightarrow \text{Label} \\
\text{handler} [\] l T = \text{arbitrary} \\
\text{handler} (x\#xs) l T = \text{if } (x.\text{start} \leq l < x.\text{end} \land T \preceq x.\text{excType}) \text{ then } (x.\text{target}) \text{ else } (\text{handler } xs l T) \\
\]

\textbf{Rules}

In the following, we present the rules for \texttt{throw} instruction, and .try catch and .try .finally structures. Furthermore, we extent Bannwart and Müller’s method call rule \cite{6} to handle exceptions.

\textbf{Throw.} To prove the instruction specification \{ \ P \ \} \ l: \texttt{throw}, we need to show that the \( P \) implies the precondition of the instruction where the exception is caught. This precondition is obtained using the function \textit{handler}. To simplify the notation, we write \textit{handler}(l) for the application of the function \textit{handler} with the exception table constructed from the current CIL program, the label \( l \), and the type of the current exception value. The rule for \texttt{throw} is defined as follows:
5.2 A Bytecode Logic for CIL

\[
P \Rightarrow E_{\text{handler}(l)} [s(0)/excV]
\]
\[
\mathcal{A} \vdash \{ P \} l : \text{throw}
\]

Similar to the source logic, we use the variable \( excV \) to store the current exception value. The instruction \texttt{throw} takes the top of the stack (an object of type \texttt{Exception}), and assigns it to \( excV \). Then, control is transferred to the label where the exception is caught.

In the implication \( P \Rightarrow E_{\text{handler}(l)} [s(0)/excV] \), the replacement \([s(0)/excV]\) is used to assign the top of the stack to \( excV \). Note that the instruction \texttt{throw} does not pop the exception; after control is transferred, the exception is still on top of the stack.

Try-catch. The body of the .\texttt{try} block as well as the body of the .\texttt{catch} block is a sequence of instruction specifications. The precondition of the .\texttt{try} instruction is defined by the precondition of the first instruction in the .\texttt{try} block. In a similar way, the precondition of the .\texttt{catch} block is defined by the precondition of the first instruction of the .\texttt{catch} block.

To show that a .\texttt{try catch} block is valid, we need to show that all the instructions in the .\texttt{try} and the .\texttt{catch} blocks are valid.

Let \( seq_1 \) and \( seq_2 \) be a list of instruction specifications, the rule is the following:

\[
\forall \text{InstSpec} \in seq_1 + seq_2 : \mathcal{A} \vdash \text{InstSpec}
\]
\[
\mathcal{A} \vdash .\text{try} \{ seq_1 \} .\text{catch} T \{ seq_2 \}
\]

Try-finally. The semantics of the .\texttt{try .finally} structure is as follows. First, the instructions of the .\texttt{try} block are executed. If any of these instructions triggers an exception, control is transferred to the finally block, and after its execution the exception is propagated. If there is no exception, then the .\texttt{try} block terminates with a \texttt{leave l} instruction. Thus, control is transferred to the finally block, and after its execution control is transferred to the label \( l \). In both cases, if the .\texttt{finally} block triggers an exception, control is transfer to the place where the exception is caught.

The .\texttt{try} block is a sequence of CIL instructions. This sequence can contain several \texttt{leave} instructions, and we assume that the last instruction of the block is a \texttt{leave} instruction (thus, there is at least one \texttt{leave} instruction). The .\texttt{finally} block is also a sequence of CIL instruction. The last instruction of the .\texttt{finally} is a \texttt{endfinally} instruction. To show that the .\texttt{try .finally} is valid, we need to show the following:

1. All instructions in the .\texttt{try} and .\texttt{finally} blocks are valid (except for the \texttt{leave} instructions in the .\texttt{try} block).

2. For all instruction specifications \{ P \} l : \texttt{leave k} in the .\texttt{try} block, the following holds:

   • the precondition of the \texttt{leave} implies the precondition of the first instruction in the .\texttt{finally} block.
• If the exception value is null, the precondition of the *endfinally* instruction implies the precondition at the label *k*.

3. If the exception value is not null, the precondition of the *endfinally* instruction implies the precondition where the exception is caught.

Let *seq*₁ and *seq*₂ be lists of instruction specifications defined as follows:

\[
\begin{align*}
*seq*₁ &= \{P₁\}l₁ : I₁ \ldots \{Pᵢ\}lᵢ : leave lᵢₙ \\
*seq*₂ &= \{Pⱼ\}lⱼ : Iⱼ \ldots \{Pₖ\}lₖ : endfinally
\end{align*}
\]

Let *seq*' denote all the CIL instructions in *seq*₁ except for the *leave* instructions. The rule is defined as follows:

\[
\forall \text{InstSpec} \in \text{*seq*}'₁ + \text{*seq*}₂ : \mathcal{A} \vdash \text{InstSpec}
\]

for all *leave* instructions \(\in \text{*seq*}₁ \) of the form \(\{Pᵢ\}lᵢ : leave lᵢₙ \) then \(Pᵢ \Rightarrow Pⱼ \) and \(excV = \text{null} \land Pₖ \Rightarrow Eᵢₙ \)

\[
\begin{align*}
&\quad excV \neq \text{null} \land Pₖ \Rightarrow \text{handler}(lᵢ) \\
\Rightarrow \mathcal{A} \vdash .\text{try} \{ \text{*seq*}₁ \} .\text{finally} \{ \text{*seq*}₂ \}
\end{align*}
\]

The rule uses the sequence *seq*' instead of *seq*₁ because the *leave* instructions in the *try* block of a *try* *finally* instruction have a special semantics. The precondition of the *leave* has to imply the precondition of the first instruction of the *finally* block (if the *leave* instruction is used outside a *try* *finally* instruction, the precondition of the *leave* implies the precondition of its target label).

**Method Calls.** In this section, we extend Bannwart and Müller’s method call rule [6] to handle exceptions. To prove the call of a virtual routine *T:*m, one has to show:

1. *T:*m satisfies its method specification \(\mathcal{A} \vdash \{P\} \). *T:*m \(\{Qₙ, Qₑ\}\),

2. If \(s(1)\) is not null, the precondition of the callvirt instruction, \(Eᵢ\), implies the precondition of the method specification, \(P\), with actual arguments replaced by the formal parameters,

3. If \(s(1)\) is null, the precondition of the callvirt instruction, \(Eᵢ\), implies the precondition where the null reference exception is caught,

4. If *T:*m does not trigger an exception (\(excV\) is null), the normal postcondition, \(Qₙ\), implies the precondition of the successor instruction, \(Eᵢ₊₁\), with \(Result\) replaced by top of the stack, and
5. If \( T: m \) triggers an exception (\( \text{excV} \) is not null), the exceptional postcondition, \( Q_e \), implies the precondition at the label where the exception is caught.

To simplify the rule, the function \( \text{handler} \) does not take the current exception table explicitly. The rule is defined as follows:

\[
\mathcal{A} \vdash \{ P \} \quad T: m \quad \{ Q_n , Q_e \}
\]

\[
s(1) \neq \text{null} \land E_l \Rightarrow P[s(1)/\text{this}, s(0)/p][\text{shift}(w)/Z] \\
s(1) = \text{null} \land E_l \Rightarrow E_{\text{handler}}(l, \text{NullReference})
\]

\[
\text{excV} = \text{null} \land Q_n[s(0)/\text{result}][w/Z] \Rightarrow E_{l+1} \\
\text{excV} \neq \text{null} \land Q_e[w/z] \Rightarrow E_{\text{handler}}(l, \tau(\text{excV}))
\]

\[
\mathcal{A} \vdash \{ E_l \} \quad l : \text{callvirt} \quad T: m
\]

cil invocation rule

where \( Z \) is a vector \( Z_0, ..., Z_n \) of logical variables and \( w \) is a vector \( w_0, ..., w_n \) of local variables or stack elements (different from \( s(0) \)).

A method call does not modify the local variables and the evaluation stack of the caller, except for popping the arguments and pushing the result of the call. To express these frame properties, the invocation rule allows one to substitute logical variables in the methods pre- and postcondition by local variables and stack elements of the caller. However, \( s(0) \) must not be used for a substitution because it contains the result of the call, that is, its value is not preserved by the call.

5.2.4 Example

In the following, we present part of the bytecode proof of the example in Figure 5.1. We have applied the bytecode logic to the from body of the loop (lines \( IL005 - IL009 \)). To make the proof more interesting, we change the compiled bytecode. First, the constants 2 and 1 are pushed, and then the values are popped to the variables \( i \) and \( \text{Result} \). The complete proof of example 5.1 is presented in Section 6.6. The bytecode specifications are the following:

\[
\{ n > 1 \} \quad IL005 : \text{ldc} \ 2 \\
\{ n > 1 \land s(0) = 2 \} \quad IL006 : \text{ldc} \ 1 \\
\{ n > 1 \land s(0) = 1 \land s(1) = 2 \} \quad IL007 : \text{stloc} \ \text{Result} \\
\{ n > 1 \land \text{Result} = 1 \land s(0) = 2 \} \quad IL008 : \text{stloc} \ i \\
\{ n > 1 \land \text{Result} = 1 \land i = 2 \} \quad IL009 : \text{br} \ \text{IL020}
\]

To be able to show that the bytecode specifications at labels \( IL005 - IL008 \) are valid, we need to show:
for all \( l : \{ IL005, \ldots, IL008 \} : \quad E_l \Rightarrow wp(l) \\
\vdash \{ E_l \} \ l : I_l \)

We prove it applying the \( wp \) definition. In the following, we present this proof.

• Case \( IL005 : \quad (n > 1) \Rightarrow wp(ldc 2) \\
\vdash \{ n > 1 \} \ \text{IL005} : \ ldc 2 \)

Applying the \( wp \) definition for \( ldc \), we need to prove:

\[ (n > 1) \Rightarrow unshift((n > 1 \land s(0) = 2)[2/s(0)]) \]

We prove this implication as follows:

\[ (n > 1) \Rightarrow unshift((n > 1 \land s(0) = 2)[2/s(0)]) \]

[definition of replacement]

\[ (n > 1) \Rightarrow unshift(n > 1 \land 2 = 2) \]

[definition of \textit{unshift}]

\[ (n > 1) \Rightarrow (n > 1 \land 2 = 2) \]

□

• Case \( IL006 : \quad (n > 1 \land s(0) = 2) \Rightarrow wp(ldc 1) \\
\vdash \{ n > 1 \land s(0) = 2 \} \ \text{IL006} : \ ldc 1 \)

By the definition of \( wp \) for \( ldc \), we need to prove:

\[ (n > 1 \land s(0) = 2) \Rightarrow unshift((n > 1 \land s(0) = 1 \land s(1) = 2)[1/s(0)]) \]

We show that this implication holds applying the definitions of replacement and \textit{unshift}:

\[ (n > 1 \land s(0) = 2) \Rightarrow unshift((n > 1 \land s(0) = 1 \land s(1) = 2)[1/s(0)]) \]

[definition of replacement]

\[ (n > 1 \land s(0) = 2) \Rightarrow unshift(n > 1 \land 1 = 1 \land s(1) = 2) \]

[definition of \textit{unshift}]

\[ (n > 1 \land s(0) = 2) \Rightarrow (n > 1 \land 1 = 1 \land s(0) = 2) \]

□
5.2 A Bytecode Logic for CIL

**Case IL007:** \( (n > 1 \land s(0) = 1 \land s(1) = 2) \Rightarrow wp(stloc \ Result) \)

\[ \vdash \{ n > 1 \land s(0) = 1 \land s(1) = 2 \} \quad IL007 : \ stloc \ Result \]

By the definition of \( wp \) for \( stloc \), we need to show:

\( (n > 1 \land s(0) = 1 \land s(1) = 2) \Rightarrow shift(n > 1 \land Result = 1 \land s(0) = 2) \ [s(0)/Result] \)

We prove this implication in the following way:

\( (n > 1 \land s(0) = 1 \land s(1) = 2) \Rightarrow shift(n > 1 \land Result = 1 \land s(0) = 2) \ [s(0)/Result] \)

[definition of shift]

\( (n > 1 \land s(0) = 1 \land s(1) = 2) \Rightarrow (n > 1 \land Result = 1 \land s(1) = 2) \ [s(0)/Result] \)

[definition of replacement]

\( (n > 1 \land s(0) = 1 \land s(1) = 2) \Rightarrow (n > 1 \land s(0) = 1 \land s(1) = 2) \)

\( \square \)

**Case IL008:** \( (n > 1 \land Result = 1 \land s(0) = 2) \Rightarrow wp(stloc \ i) \)

\[ \vdash \{ n > 1 \land Result = 1 \land s(0) = 2 \} \quad IL008 : \ stloc \ i \]

By the definition of \( wp \) for \( stloc \), we need to show:

\( (n > 1 \land Result = 1 \land s(0) = 2) \Rightarrow shift(n > 1 \land Result = 1 \land i = 2) \ [s(0)/i] \)

We prove it as follows:

\( (n > 1 \land Result = 1 \land s(0) = 2) \Rightarrow shift(n > 1 \land Result = 1 \land i = 2) \ [s(0)/i] \)

[definition of shift]

\( (n > 1 \land Result = 1 \land s(0) = 2) \Rightarrow (n > 1 \land Result = 1 \land i = 2) \ [s(0)/i] \)

[definition of replacement]

\( (n > 1 \land Result = 1 \land s(0) = 2) \Rightarrow (n > 1 \land Result = 1 \land s(0) = 2) \)

\( \square \)
Chapter 6

Proof-Transforming Compilation for the Core Language

Our proof-transforming compiler translates Hoare-style proofs of source programs into bytecode proofs. This chapter presents a proof-transforming compiler for the core object-oriented language. The compiler is based on four transformation functions: one function \( \nabla_P \) to translate the source proof, one function, \( \nabla_B \) to translate a proof tree, one function \( \nabla_S \) for instructions, and one for expressions \( \nabla_E \). The function \( \nabla_P \) yields a list of CIL proofs (a derivation in the bytecode logic); and the function \( \nabla_B \) yields a CIL proof. The functions \( \nabla_E \) and \( \nabla_S \) yield a sequence of CIL bytecode instructions and their specification.

This chapter is organized as follows. First, we present the basics for the translation. Second, we present the translation of virtual routine and routine implementations, the translation of expressions, the translation of instructions, and the translation of language-independent rules. Finally, the chapter shows an application of the proof transformation, and it concludes with the soundness theorem and related work.

This chapter is based on the technical report [81].

6.1 Translation Basics

The proof of the source program consists of a list of proof trees, where a proof tree is a derivation in the source logic as defined in Chapter [3] Section [3.1]. The definition is:

\[
\text{datatype } \text{Proof} = \text{list_of } \text{ProofTree}
\]

Bytecode proofs consist of a list of CIL proof trees, where a CIL proof tree is a derivation in the CIL logic as defined in Chapter [5]. This derivation is the application of the CIL method specification rules such as the \textit{routine implementation rule}, the \textit{class rule}, and the language independent rules. The leaves of the CIL proof trees are bytecode proofs.
Proof-Transforming Compilation for the Core Language

(a list of instruction specifications). These leaves connect the CIL method specification
rules and the instruction specifications. The connection is done by applying the cil method
body rule (Section 5.2.2). An instruction specification consists of a precondition, a label,
and a CIL intruction. The definition is:

```
datatype CILProof = list_of CILProofTree
datatype BytecodeProof = list_of InstrSpec
datatype InstrSpec = Precondition Label Instruction
```

Translating Source Proofs. The main translation function \( \nabla_P \) takes a source proof,
and yields a CIL proof. This function is defined using the function \( \nabla_B \), which translates
the source proof tree. To be able to translate the source proof, we use the function \( \nabla_S \)
for the translation of instructions, and the function \( \nabla_E \) for expressions. The signatures of
these functions are defined as follows:

\[
\begin{align*}
\nabla_P &: \text{Proof} \to \text{CILProof} \\
\nabla_B &: \text{ProofTree} \to \text{CILProofTree} \\
\nabla_E &: \text{Precondition} \times \text{Expression} \times \text{Postcondition} \times \text{Label} \to \text{BytecodeProof} \\
\nabla_S &: \text{ProofTree} \times \text{Label} \times \text{Label} \times \text{Label} \to \text{BytecodeProof}
\end{align*}
\]

The definitions of the function \( \nabla_P \) is straightforward. Given a source proof \( p \) defined
as the list \( tree_1, ..., tree_n \), where \( tree_i \) is a proof tree in the source logic, the function \( \nabla_P \)
is defined as follows:

\[
\nabla_P (p) = \nabla_B(tree_1) + ... + \nabla_B(tree_n)
\]

The definition of the function \( \nabla_B \) is presented in the following section. The functions
\( \nabla_E \) and \( \nabla_S \) are defined in Section 6.3 and Section 6.4, respectively.

6.2 Proof Translation of Routines

The logic for the source language and the logic for bytecode treat routine specifications
in the same way: using virtual routines \( T:m \) and routine implementations \( T@m \). In the
bytecode logic, the rules for method specifications are similar to the rules in the logic of
the source language. Furthermore, the bytecode logic has language-independent rules
that can be applied to method specifications. This treatment of routine specifications makes
the proof transformation simpler.
The translation of virtual routines and routine implementations are performed using the function $\nabla_B$. This function takes a proof tree of a routine specification (a derivation in the source logic) and produces a CIL proof tree (a derivation in the CIL logic). Thus, we define the function $\nabla_B$ for all rules that allows proving routine specifications. These rules are the class rule, subtype rule, routine implementation rule, and the language-independent rules. Following, we present the translation of these rules.

### 6.2.1 Class Rule

Let $T_{imp}$ and $T_{tm}$ be the following proof trees:

\[
T_{imp} \equiv \frac{\mathcal{A} \vdash \{ \tau(\text{Current}) = T \land P \} \quad \text{imp}(T, m) \quad \{ Q_n, Q_e \}}{\quad \text{Tree}_1}
\]

\[
T_{tm} \equiv \frac{\mathcal{A} \vdash \{ \tau(\text{Current}) < T \land P \} \quad T:m \quad \{ Q_n, Q_e \}}{\quad \text{Tree}_2}
\]

where $\text{Tree}_1$ and $\text{Tree}_2$ are the derivations used to prove the Hoare triples of $\text{imp}(T, m)$ and $T:m$ respectively.

The definition of the translation is the following:

\[
\nabla_B \left( \frac{\mathcal{A} \vdash \{ \tau(\text{Current}) \leq T \land P \} \quad T_{imp} \quad T_{tm} \quad T:m \quad \{ Q_n, Q_e \}}{\quad \text{cil class rule}} \right) =
\]

\[
\nabla_B( T_{imp} ) \quad \nabla_B( T_{tm} )
\]

Note that the translation of the class rule produces a derivation in the logic for CIL that applies the CIL class rule. The hypothesis of this CIL rule is the translation of the proof trees $T_{imp}$ and $T_{tm}$. To produce this translation, we apply the translation function $\nabla_B$ to $T_{imp}$ and $T_{tm}$.

### 6.2.2 Subtype Rule

Let $T_{sm}$ be the following proof tree:
The definition of the translation of the subtype rule is the following:

\[
\nabla_B \left( \frac{S \preceq T \quad T_{sm}}{A \vdash \{ \tau(\text{Current}) \preceq S \land P \} \quad T \vdash m \quad \{ Q_n, Q_e \}} \right) =
\]

\[
\frac{S \preceq T \quad \nabla_B (T_{sm})}{A \vdash \{ \tau(\text{Current}) \preceq S \land P \} \quad T \vdash m \quad \{ Q_n, Q_e \}}
\]

CIL subtype rule

The translation of the subtype rule produces the CIL derivation that applies the CIL subtype rule. The hypothesis of this derivation is the translation of the proof tree \(T_{sm}\) and the expression \(S \preceq T\). The expression \(S \preceq T\) does not need to be translated because the class structures of the source and the bytecode are the same. (The CIL subtyping function is also denoted with \(\preceq\)).

### 6.2.3 Routine Implementation Rule

Let \(T_{body}\) be the following proof tree:

\[
T_{body} \equiv \frac{\text{Tree}_1}{A, \{P\} \quad T \vdash m \quad \{ Q_n, Q_e \} \vdash \{ P \} \quad \text{body}(T \vdash m) \quad \{ Q_n, Q_e \}}
\]

The translation is defined as follows:

\[
\nabla_B \left( \frac{T_{body}}{A \vdash \{ P \} \quad T \vdash m \quad \{ Q_n, Q_e \}} \right) =
\]

\[
\frac{\nabla_S(T_{body})}{A, \{P\} T \vdash m \{ Q_n, Q_e \} \vdash \{ P \} \quad \text{body}_{cil}(T \vdash m) \quad \{ Q_n, Q_e \}}
\]

CIL rule

\[
\frac{\nabla_B(T_{body})}{A \vdash \{ P \} \quad T \vdash m \quad \{ Q_n, Q_e \}}
\]

CIL r. impl. rule
This translation first applies the CIL routine implementation rule. The hypothesis of this rule application is the application of the CIL body rule. The body rule allows connecting the CIL method specification and the CIL instruction specification. The hypothesis of the body rule is a list of bytecode specifications produced by the translation function $\nabla_S$. The function $\nabla_S$ is defined in Section 6.4.

### 6.2.4 Language-Independent Rules for Routines

In the logic for the source language, languages-independent rules are applied to both routine specifications and instructions specifications. We translate these rules in two parts: the translation of language-independent rules for routine specifications and the translation for instructions. These two translation are needed because they produce different results. The translation of routine specifications produces a derivation in the bytecode language, and the translation of instructions produces a list of bytecode specifications. In this section, we present the translation of language-independent rules for routine specifications; Section 6.5 presents the translation of language-independent rules for instructions.

To translate these rules for routine specification, we use the language-independent rules of the bytecode logic.

**Assumpt-axiom**

The assumpt axiom in the logic for the source language is translated using the assumpt axiom of the bytecode logic as follows:

$$\nabla_B \left( \frac{A \vdash A}{A \vdash A} \right) = \frac{A \vdash A}{A \vdash A} \text{ cil assumpt axiom}$$

**False axiom**

To translate the false axiom, we use the false axiom of the bytecode logic. The translation is:

$$\nabla_B \left( \frac{\vdash \{ \text{false} \} \quad s_1 \quad \{ \text{false} \), \text{false} \}}{\vdash \{ \text{false} \} \quad s_1 \quad \{ \text{false} \), \text{false} \}} \right) =$$

$$\frac{\vdash \{ \text{false} \} \quad s_1 \quad \{ \text{false} \), \text{false} \}}{\vdash \{ \text{false} \} \quad s_1 \quad \{ \text{false} \), \text{false} \}} \text{ cil false axiom}$$

**Assumpt-intro-axiom**

Let $T_A$ be the following proof tree:
\[ T_A \equiv \frac{\text{Tree}_1}{\mathcal{A} \vdash A} \]

The assumpt-intro-axiom in the logic of the source program is mapped to the assumpt-intro-axiom of bytecode logic using as hypothesis the translation of the hypothesis of the source triple:

\[ \nabla_B \left( \frac{T_A}{A_0, \mathcal{A} \vdash A} \right) = \frac{\nabla_B (T_A)}{A_0, \mathcal{A} \vdash A} \text{ cil assumpt-intro-axiom} \]

**Assumpt-elim-axiom**

Let \( T_A \) and \( T_{A_0} \) be the following proof trees:

\[ T_A \equiv \frac{\text{Tree}_1}{\mathcal{A} \vdash A_0} \quad T_{A_0} \equiv \frac{\text{Tree}_2}{A_0, \mathcal{A} \vdash A} \]

The assumpt-elim-axiom is translated using the CIL assumpt-elim-axiom where the hypothesis are the translations of the hypotheses in the source rule. This translation is defined as follows:

\[ \nabla_B \left( \frac{T_A}{A_0, \mathcal{A} \vdash A} \right) = \frac{\nabla_B (T_A)}{A \vdash A} \frac{\nabla_B (T_{A_0})}{A \vdash A} \text{ cil assumpt-elim-axiom} \]

**Strength**

Let \( T_{S_1} \) be the following proof tree:

\[ T_{S_1} \equiv \frac{\text{Tree}_1}{\mathcal{A} \vdash \{ P \}} \frac{\text{Tree}_e}{s_1 \{ Q_n, Q_e \}} \]

The definition is:

\[ \nabla_B \left( \frac{P' \Rightarrow P}{\mathcal{A} \vdash \{ P' \}} \frac{T_{S_1}}{s_1 \{ Q_n, Q_e \}} \right) = \frac{P' \Rightarrow P}{\mathcal{A} \vdash \{ P' \}} \frac{\nabla_B (T_{S_1})}{s_1 \{ Q_n, Q_e \}} \text{ cil strength rule} \]

This translation uses the CIL strength rule. We assume that the implication \( P' \Rightarrow P \) is proved in a theorem prover. Since the translation does not modify the expressions \( P' \) and \( P \), the same proof is used in the bytecode proof. We assume that the proof checker verifies the proof with the same theorem prover used to develop the source proof. These assumptions simplify the translation.
6.2 Proof Translation of Routines

Weak

Let $T_{S_1}$ be the following proof tree:

$$T_{S_1} \equiv \frac{\text{Tree}_1}{\mathcal{A} \vdash \{ P \} \quad s_1 \quad \{ Q_n', Q_e' \}}$$

The translation is defined as following:

$$\nabla_B \left( \frac{T_{S_1} \quad Q_n \Rightarrow Q_n' \quad Q_e \Rightarrow Q_e'}{\mathcal{A} \vdash \{ P \} \quad s_1 \quad \{ Q_n', Q_e' \}} \right) =$$

$$\nabla_B \left( T_{S_1} \right) \quad Q_n \Rightarrow Q_n' \quad Q_e \Rightarrow Q_e'$$

cil weak rule

Similar to the translation of the strength rule, the proofs of implications $Q_n \Rightarrow Q_n'$ and $Q_e \Rightarrow Q_e'$.

Conjunction

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

$$T_{S_1} \equiv \frac{\text{Tree}_1}{\mathcal{A} \vdash \{ P' \} \quad s_1 \quad \{ Q_n', Q_e' \}} \quad T_{S_2} \equiv \frac{\text{Tree}_2}{\mathcal{A} \vdash \{ P'' \} \quad s_1 \quad \{ Q_n'', Q_e'' \}}$$

The translation of the conjunction rule yields the application of the CIL conjunction rule. The definition is:

$$\nabla_B \left( \frac{T_{S_1} \quad T_{S_2}}{\mathcal{A} \vdash \{ P' \land P'' \} \quad s_1 \quad \{ Q_n' \land Q_n'', Q_e' \land Q_e'' \}} \right) =$$

$$\nabla_B \left( T_{S_1} \right) \quad \nabla_B \left( T_{S_2} \right)$$

cil conjunction rule

Disjunction

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

$$T_{S_1} \equiv \frac{\text{Tree}_1}{\mathcal{A} \vdash \{ P' \} \quad s_1 \quad \{ Q_n', Q_e' \}} \quad T_{S_2} \equiv \frac{\text{Tree}_2}{\mathcal{A} \vdash \{ P'' \} \quad s_1 \quad \{ Q_n'', Q_e'' \}}$$
Similar to the conjunction rule, this translation yields the application of the CIL disjunction rule. The definition is:

$$\nabla_B \left( \frac{T_{S_1} \quad T_{S_2}}{\mathcal{A} \vdash \{ P' \lor P'' \} \quad s_1 \quad \{ Q'_n \lor Q''_n \lor Q'_e \lor Q''_e \}} \right) =$$

$$\nabla_B (T_{S_1}) \quad \nabla_B (T_{S_2})$$

\text{cil disjunction rule}

**Invariant**

Let $T_{S_1}$ be the following proof tree:

$$T_{S_1} \equiv \frac{\text{Tree}_1}{\mathcal{A} \vdash \{ P \} \quad s_1 \quad \{ Q_n \lor Q_e \}}$$

The translation is defined as follows:

$$\nabla_B \left( \frac{T_{S_1}}{\mathcal{A} \vdash \{ P \land W \} \quad s_1 \quad \{ Q_n \land W \lor Q_e \land W \}} \right) =$$

$$\nabla_B (T_{S_1})$$

\text{cil invariant rule}

**Substitution**

Let $T_{S_1}$ be the following proof tree:

$$T_{S_1} \equiv \frac{\text{Tree}_1}{\mathcal{A} \vdash \{ P \} \quad s_1 \quad \{ Q_n \lor Q_e \}}$$

The translation is the following:

$$\nabla_B \left( \frac{T_{S_1}}{\mathcal{A} \vdash \{ P[t/Z] \} \quad s_1 \quad \{ Q_n[t/Z] \lor Q_e[t/Z] \}} \right) =$$

$$\nabla_B (T_{S_1})$$

\text{cil substitution rule}
6.3 Proof Translation of Expressions

All-rule

Let $T_{S_1}$ be the following proof tree:

$$T_{S_1} \equiv \frac{Tree_1}{A \vdash \{ P[Y/Z] \} \ s_1 \ \{ Q_n \ , \ Q_e \}}$$

The translation is:

$$\nabla_B \left( \frac{T_{S_1}}{A \vdash \{ P[Y/Z] \} \ s_1 \ \{ \forall Z : Q_n \ , \ \forall Z : Q_e \}} \right) =$$

$$\nabla_B \left( T_{S_1} \right)$$

$$\frac{A \vdash \{ P[Y/Z] \} \ s_1 \ \{ \forall Z : Q_n \ , \ \forall Z : Q_e \}}{A \vdash \{ P[Y/Z] \} \ s_1 \ \{ \forall Z : Q_n \ , \ \forall Z : Q_e \}}$$

Ex-rule

Let $T_{S_1}$ be the following proof tree:

$$T_{S_1} \equiv \frac{Tree_1}{A \vdash \{ P \} \ s_1 \ \{ Q_n[Y/Z] \ , \ Q_e[Y/Z] \}}$$

The translation is:

$$\nabla_B \left( \frac{T_{S_1}}{A \vdash \{ \exists Z : P \} \ s_1 \ \{ Q_n[Y/Z] \ , \ Q_e[Y/Z] \}} \right) =$$

$$\nabla_B \left( T_{S_1} \right)$$

$$\frac{A \vdash \{ \exists Z : P \} \ s_1 \ \{ Q_n[Y/Z] \ , \ Q_e[Y/Z] \}}{A \vdash \{ \exists Z : P \} \ s_1 \ \{ Q_n[Y/Z] \ , \ Q_e[Y/Z] \}}$$

6.3 Proof Translation of Expressions

Expressions are translated using the function $\nabla_E$. This function generates a bytecode proof from a source expression and a precondition for its evaluation. This function is defined as a composition of the translations of the subexpressions. The signature is:

$$\nabla_E : \text{Precondition} \times \text{Expression} \times \text{Postcondition} \times \text{Label} \rightarrow \text{BytecodeProof}$$

In $\nabla_E$ the label is used as the starting label of the translation. Following, we present the translation of constants, variables, unary and binary expressions. To simplify the translation, we do not distinguish between arguments and local variables. Thus, the translation uses $\text{stloc}$ or $\text{ldloc}$ instead of $\text{starg}$ and $\text{ldarg}$ when the variable is an argument.
6.3.1 Constants

Constants are translated using the \texttt{ldc} instruction. The constant is loaded on top of the stack. The precondition of the instruction specification is \( Q \land \text{unshift}(P[c/s(0)]) \). The translation is defined as follows:

\[
\nabla_E( Q \land \text{unshift}(P[c/s(0)]) \), \ c \ , \ shift(Q) \land P \ , \ l_{\text{start}} ) =
\{ Q \land \text{unshift}(P[c/s(0)]) \} \quad l_{\text{start}} : \text{ldc} \ c
\]

Although, the PTC is not part of the trusted computing base, it is interesting to prove soundness. Soundness means that the compiler produces a valid proof. To show that the translation is sound one needs to show that the precondition at label \( l_{\text{start}} \) implies the weakest precondition of the instruction \texttt{ldc} using the postcondition \( \text{shift}(Q) \land P \). This implication holds as follows:

\[
Q \land \text{unshift}(P[c/s(0)]) \quad \text{implies} \quad \wp(\text{ldc} \ c, \text{shift}(Q) \land P)
\]
\[
\Leftrightarrow \quad [\text{definition of } \wp]
\]
\[
Q \land \text{unshift}(P[c/s(0)]) \quad \text{implies} \quad \text{unshift}( \text{shift}(Q) \land P[c/s(0)] )
\]
\[
\Leftrightarrow \quad [\text{definition of } \text{unshift}]
\]
\[
Q \land \text{unshift}(P[c/s(0)]) \quad \text{implies} \quad Q \land \text{unshift}(P[c/s(0)])
\]
\[
\square
\]

6.3.2 Variables

Similar to the translation of constants, variables are translated loading the variable on top of the stack using the instruction \texttt{ldloc}. The definition is:

\[
\nabla_E( Q \land \text{unshift}(P[x/s(0)]) \), \ x \ , \ shift(Q) \land P \ , \ l_{\text{start}} ) =
\{ Q \land \text{unshift}(P[x/s(0)]) \} \quad l_{\text{start}} : \text{ldloc} \ x
\]

6.3.3 Binary Expressions

To translate binary expressions \( e_1 \ op \ e_2 \), first the expression \( e_1 \) is translated applying the function \( \nabla_E \), then the expression \( e_2 \) is translated, and finally the binary operation is added. The translation is defined as follows:
6.4 Poof Translation of Instructions

\[ \nabla_E(Q \land \text{unshift}(P[e_1 \ op \ e_2/s(0)]), \ e_1 \ op \ e_2, \ shift(Q) \land P, \ \text{l}_{\text{start}}) = \]

\[ \nabla_E(Q \land \text{unshift}(P[e_1 \ op \ e_2/s(0)]), \ e_1, \ shift(Q) \land P[s(0) \ op \ e_2/s(0)], \ \text{l}_{\text{start}}) \]
\[ \nabla_E(shift(Q) \land P[s(0) \ op \ e_2/s(0)], \ e_2, shift(Q) \land shift(P[s(1) \ op \ s(0)/s(1)], l_b) \]
\{ shift^2(Q) \land shift(P[s(1) \ op \ s(0)/s(1)]) \} \ l_c : \text{binop}_o \]

The instruction \( \text{binop} \) pops the top two values from the stack, then it applies the binary operation, and finally pushes the result on top of the stack. To obtain this translation, we have applied the weakest precondition starting with the postcondition \( shift(Q) \land P \). The definition of wp of \( \text{binop} \) is \( (shift(E_{l+1}))(s(1)ops(0))/s(1)) \), so we get:

\[ shift^2(Q) \land shift(P[s(1) \ op \ s(0)/s(1)]) \]

The precondition of the translation of \( e_2 \) is obtained replacing \( s(0) \) by \( e_2 \), and then applying \( \text{unshift} \) (this result is the application of wp definition for \( \text{idloc} \)). Finally, the precondition of \( e_1 \) is the obtained replacing \( s(0) \) by \( e_1 \), and applying the function \( \text{unshift} \).

6.3.4 Unary Expressions

Unary expressions are translated in a similar way to binary operations. The subexpression \( e \) is translated using the translation function \( \nabla_E \). The definition is:

\[ \nabla_E(Q \land \text{unshift}(P[unop \ e/s(0)]), \ unop \ e, \ shift(Q) \land P, \ \text{l}_{\text{start}}) = \]

\[ \nabla_E(Q \land \text{unshift}(P[unop \ e/s(0)]), \ e, \ shift(Q) \land P[unop \ s(0)/s(0)], \ \text{l}_{\text{start}}) \]
\{ shift(Q) \land P[unop \ s(0)/s(0)] \} \ l_b : \text{unop}_o \]

6.4 Poof Translation of Instructions

Instructions are translated using the function \( \nabla_S \). This function takes a proof tree (a derivation in the source logic), and yields a sequence of Bytecode instructions and their specifications. The function \( \nabla_S \) is defined as a composition of the translations of the proof’s sub-trees. The signature is:
\[ \nabla_S : \text{ProofTree} \times \text{Label} \times \text{Label} \times \text{Label} \rightarrow \text{BytecodeProof} \]

In \( \nabla_S \), the three labels are: (1) \text{start} for the first label of the resulting bytecode; (2) \text{next} for the label after the resulting bytecode; this is for instance used in the translation of an \text{else} branch to determine where to jump at the end; (3) \text{exc} for the jump target when an exception is thrown. In this chapter, the label \text{exc} always refer to the last instruction of the routine since the source language does not contain instructions for catching exceptions. The translation is extended in Chapter \(\overset{[7]}{\text{?}}\) with the translation of \text{rescue} clauses, and in Chapter \(\overset{[8]}{\text{?}}\) with the translation of \text{try-catch}, \text{try-finally}, and \text{throw} instructions.

### 6.4.1 Assignment Axiom

In the assignment translation, first the expression \(e\) is translated using the function \(\nabla_E\). Then, the result is stored to \(x\) using \text{stloc}. The definition of the translation is the following:

\[
\nabla_S \left( \left. \begin{array}{c}
\mathcal{A} \vdash \left\{ \begin{array}{l}
(safe(e) \land P[e/x]) \lor \\
(\neg safe(e) \land Q_e)
\end{array} \right. \right\}
\end{array}, l_{\text{start}}, l_{\text{next}}, l_{\text{exc}} \right) = \left. \begin{array}{c}
x := e \left\{ P, Q_e \right\} \end{array} \right. , \{ \text{shift}(safe(e) \land P[e/x] \land s(0) = e), l_{\text{start}} \}
\]

\[
\nabla_E \left( \left. \begin{array}{c}
(safe(e) \land P[e/x]) \lor \\
(\neg safe(e) \land Q_e)
\end{array} \right. \right), e, \left. \begin{array}{c}
(\text{shift}(safe(e) \land P[e/x]) \land s(0) = e) \end{array} \right. , l_{\text{start}} \right)
\]

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

The precondition of the application of the function \(\nabla_E\) is the precondition of the assignment rule. The postcondition of application of the function \(\nabla_E\) is

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

because if \(e\) triggers an exception, control is transferred to the end of the routine. The precondition of the assignment is equal to the precondition of application of \(\nabla_E\). Also, the postcondition of \(\nabla_E\) implies the precondition at the label \(l_b\). Applying the definition of the weakest precondition of \text{stloc} \; x to \(P\) we can show that the instruction specification at \(l_b\) is valid:
shift(\text{safe}(e) \land P[e/x]) \land s(0) = e \implies wp(\text{stloc } x, P)
\Leftrightarrow \quad \text{[definition of wp]}
shift(\text{safe}(e) \land P[e/x]) \land s(0) = e \implies \text{shift}(P[s(0)/x])
\square

Therefore, the translation produces a sequence of valid instruction specifications. Section 6.7 discusses soundness.

6.4.2 Compound Rule

Compound instructions are the simplest instructions to translate. The translation of \(s_2\) is added after the translation of \(s_1\) where the starting label is updated to \(l_b\). Let \(T_{S_1}\) and \(T_{S_2}\) be the following proof trees:

\[
\begin{align*}
T_{S_1} &\equiv \left\{ P \right\} \quad \text{Tree}_1 \quad \left\{ Q_n, R_e \right\} \\
T_{S_2} &\equiv \left\{ Q_n \right\} \quad \text{Tree}_2 \quad \left\{ R_n, R_e \right\}
\end{align*}
\]

The definition of the translation is the following:

\[
\nabla_S \left( \left\{ P \right\} \quad T_{S_1} \quad T_{S_2} \quad \left\{ R_n, R_e \right\}, \quad l_{\text{start}}, l_{\text{next}}, l_{\text{exc}} \right) =
\n\nabla_S \left( T_{S_1}, \quad l_{\text{start}}, l_b, l_{\text{exc}} \right)
\n\n\nabla_S \left( T_{S_2}, \quad l_b, l_{\text{next}}, l_{\text{exc}} \right)
\]

The bytecode for \(s_1\) establishes \(Q_n\), which is the precondition of the first instruction of the bytecode for \(s_2\). Therefore, the concatenation of the bytecode as the result of the translation of \(s_1\) and \(s_2\) produces a sequence of valid instruction specifications.

6.4.3 Conditional Rule

Let \(T_{S_1}\) and \(T_{S_2}\) be the following proof trees:
The translation is defined as follows:

\[
\begin{align*}
T_{S_1} & \equiv \frac{T_{\text{tree}_1}}{\mathcal{A} \vdash \{ P \land e \} \quad s_1 \quad \{ Q_n, Q_e \}} \\
T_{S_2} & \equiv \frac{T_{\text{tree}_2}}{\mathcal{A} \vdash \{ P \land \neg e \} \quad s_2 \quad \{ Q_n, Q_e \}}
\end{align*}
\]

In this translation, the expression of the conditional is translated using \( \nabla_S \). After the translation \( \nabla_E \), the expression is on top of the stack (thus, \( s(0) = e \)). If \( e \) is true, control is transferred to \( l_e \), and the translation of \( s_1 \) is obtained using \( \nabla_S \). Otherwise, \( s_2 \) is translated and control is transferred to the next instruction. Here, the argument \( l_{\text{next}} \) is used to obtain the label of the next instruction. This argument avoids the generation of \texttt{nop} instructions at the end of the translation.

The translation of the expression \( e \) establishes \( (\text{shift}(P) \land s(0) = e) \) because the evaluation of the expression pushes the result on top of the stack. This postcondition implies the precondition of the successor instruction \( l_b \). If the top of the stack evaluates to true, then the precondition at \( l_b \) implies the precondition of \( s_1 \) since \( e = \text{true} \). If the top of the stack evaluates to false, then the precondition at \( l_b \) implies the precondition of \( s_2 \). Both postconditions of \( l_c \) and \( l_e \) implies \( Q_n \). Therefore, the produced bytecode proof is valid.
6.4.4 Check Axiom

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

$$
\nabla_S \left( \frac{A \vdash \{ P \}}{\text{check } e \text{ end } \{ (P \land e), (P \land \neg e) \}}, l_{\text{start}}, l_{\text{next}}, l_{\text{exc}} \right) =
$$

$$
\nabla_E (P, e, \{ \text{shift}(P) \land s(0) = e \}, l_{\text{start}})
$$

\begin{align*}
\{ \text{shift}(P) \land s(0) = e \} & \quad l_b : \text{brtrue } l_{\text{next}} \\
\{ P \land \neg e \} & \quad l_c : \text{newobj } \text{Exception}() \\
\{ \text{shift}(P) \land \neg e \land s(0) = \text{new}($, \text{Exception}) \} & \quad l_d : \text{throw}
\end{align*}

The translation of the expression $e$ establishes $\text{shift}(P) \land s(0) = e$. This postcondition implies the precondition at label $l_b$. If $s(0)$ evaluates to true, then $e$ holds, and the precondition $\text{shift}(P) \land s(0) = e$ implies the normal postcondition of the check rule. If $s(0)$ evaluates to false, $\text{shift}(P) \land s(0) = e$ implies $P \land \neg e$, and the precondition at $l_b$ implies the precondition at $l_d$. Finally, $\text{shift}(P) \land \neg e \land s(0) = \text{new}($, \text{Exception}) implies the exceptional postcondition of the check rule. Therefore, the produced bytecode proof for the check rule is valid.

6.4.5 Loop Rule

Let $T_{S_1}$ be the following proof tree:

$$
T_{S_1} \equiv \frac{A \vdash \{ \neg e \land I \}}{\text{Tree}_1} s_1 \{ I, R_e \}
$$

The first step to translate the loop is to transfer control to $l_c$ where the until expression is evaluated. The body of the loop, $s_1$, is translated at label $l_b$. Finally, the instruction $\text{brfalse}$ transfer control to the instruction at label $l_b$ if the until expression is false. The definition is:
The instruction \( \text{br} \) at label \( l_{start} \) preserves \( I \), which is the precondition of the successor instruction. The translation of the expression \( e \) establishes \( \text{shift}(I) \land s(0) = e \) because the evaluation of the expression pushes the result on top of the stack. This postcondition implies the precondition of the successor instruction \( l_d \). The precondition of \( l_d \) implies both possible successor instructions, namely \( \neg e \land I \) for the successor \( l_b \) (the first instruction of the translation of \( T_{S_1} \)) and \( I \land e \) for \( l_{next} \). Finally, the translation of \( T_{S_1} \) establishes \( I \), which implies the precondition of its successor. Therefore, the produced bytecode proof is valid.

### 6.4.6 Read Attribute Axiom

Let \( P' \) be the following precondition:

\[
P' \equiv \begin{cases} 
(y \neq \text{Void} \land P[\text{instvar}(y,S@a)]/x]) \lor \\
(y = \text{Void} \land Q_e \left[ \text{null}(\text{NullPExc})/\$, \\
\text{new}(\$, \text{nullPExc})/\text{excV} \right])
\end{cases}
\]

The translation is the following:

\[
\nabla_S \left( \begin{array}{c}
A \vdash \{ P' \} \\
\text{x := y}.S@a \\
\{ P, Q_e \}
\end{array} \right), \quad l_{start}, l_{next}, l_{exc} =
\]

\[
\begin{cases} 
\{ P' \} \quad l_{start} : \text{ldloc } y \\
\{ s(0) = y \land \text{shift}(P') \} \quad l_b : \text{ldfld } S@a \\
\{ s(0) = \text{null}(y,S@a) \land \text{shift}(P[\text{null}(y,S@a)]/x]) \} \quad l_c : \text{stloc } x
\end{cases}
\]
The translation of read attribute axiom first pushes the target object \( y \) on top of the stack. Then, the value of the attribute \( a \) of the object \( y \) is obtained using the instruction \( \text{ldfld} \) (load field). This instruction pops the target object and pushes the value of the field \( S@a \) on top of the stack. Finally, this value is assigned to \( x \). If the object \( y \) is void, then an exception is triggered at label \( l_b \). The precondition of label \( l_c \) expresses that the top of stack is equal to the attribute \( S@a \) of the object \( y \). Thus, one knows that \( y \neq \text{void} \), and \( P[\$\text{instvar}(y, S@a)]/x \) holds.

### 6.4.7 Write Attribute Axiom

Let \( P' \) be the following precondition:

\[
P' \equiv \begin{cases} 
(y \neq \text{Void} \land P[\$\text{instvar}(y, S@a) := e]/$]) \lor \\
(y = \text{Void} \land Q_e \begin{bmatrix} 
\$\text{NullPExc}/$, \\
\text{new}($, \text{NullPExc})/\text{excV} 
\end{bmatrix})
\end{cases}
\]

The definition of the translation is the following:

\[

\nabla_S \left( \begin{array}{c}
\{ P' \} \\
y.S@a := e \begin{array}{c}
\{ P, \ Q_e \}
\end{array}
\end{array} , l_{start}, l_{next}, l_{exc} \right) = 
\]

\[
\nabla_E(\{s(0) = y \land \text{shift}(P')\}, \ e, \{s(1) = y \land s(0) = e \land \text{shift}^2(P')\} , l_b) \\
\{s(1) = y \land s(0) = e \land \text{shift}^2(P')\} , l_c : \text{stfld} S@a
\]

Similar to the read attribute axiom, this translation pushes the target object \( y \) and the expression \( e \) on top of the stack. The attribute \( S@a \) is updated using the instruction \( \text{stfld} \). If the target object \( y \) is void, then the instruction \( \text{stfld} \) triggers an exception.

The instruction \( \text{ldloc} \) \( y \) establishes \( s(0) = y \land \text{shift}(P') \) because it pushes \( y \) on top of the stack. The translation of the expression \( e \) establishes

\[
s(1) = y \land s(0) = e \land \text{shift}^2(P')
\]

because the evaluation of the expression pushes the result on top of the stack. This post-condition implies the precondition of the successor \( l_c \). Therefore, the produced bytecode proof is valid.
6.4.8 Routine Invocation Rule

Let $P'$ be the following precondition:

\[ P' \equiv \begin{cases} (y \neq \text{Void} \land P[y/\text{Current}, e/p]) \lor \\ (y = \text{Void} \land Q_e \left[ \frac{\$\langle \text{NullPExc} \rangle/\$$, \]
\[ \text{new}(\$, \text{NullPExc})/\text{exc}V \right] \right) \end{cases} \]

The translation is defined as follows:

\[
\nabla_E \left( \begin{array}{c} \text{Tree}_1 \\ \mathcal{A} \vdash \left\{ \begin{array}{c} P \\ T : m(p) \end{array} \right\} \left\{ \begin{array}{c} Q_n, \\ Q_e \end{array} \right\} \\ \mathcal{A} \vdash \left\{ \begin{array}{c} P' \\ x = y.T : m(e) \end{array} \right\} \left\{ \begin{array}{c} Q_n[x/\text{result}], \\ Q_e \end{array} \right\} \end{array} \right), \mathbf{l}_{\text{start}}, \mathbf{l}_{\text{next}}, \mathbf{l}_{\text{exc}} \right) = \left\{ \begin{array}{c} \mathbf{l}_{\text{start}} : \text{lldloc} \ y \\ \left\{ \begin{array}{c} \text{shift}(P') \land s(0) = y \\ e, \left\{ \text{shift}^2(P') \land s(1) = y \land s(0) = e \right\}, \mathbf{l}_b \end{array} \right\} \\ \left\{ \begin{array}{c} \text{shift}^2(P') \land s(1) = y \land s(0) = e \\ \right\} \mathbf{l}_c : \text{callvirt} \ T : m \\ \left\{ Q_n[s(0)/\text{result}] \right\} \end{array} \right\} \mathbf{l}_d : \text{stloc} \ x \right) \\
\right)
\]

The translation for routine invocations first pushes the target object on top of the stack. Second, it pushes the argument $e$. Then, it calls the routine using the callvirt instruction. Finally, it pops the result of the invocation to the variable $x$.

The most interesting part of this translation is the callvirt instruction. To show that

\[ \vdash \left\{ \text{shift}^2(P') \land s(1) = y \land s(0) = e \right\} \mathbf{l}_c : \text{callvirt} \ T : m \]

is valid, one has to apply the virtual call rule of the bytecode logic. We obtain the derivation of the virtual call rule as follows. Let $T_{tm}$ be the following proof tree:

\[ T_{tm} \equiv \frac{\text{Tree}_1}{\mathcal{A} \vdash \left\{ \begin{array}{c} P \\ T : m(p) \end{array} \right\} \left\{ \begin{array}{c} Q_n, \\ Q_e \end{array} \right\}} \]

The translation of the CIL method invocation is:
6.4 Proof Translation of Instructions

\[ \nabla_B (T_{tm}) \]

\[
s(1) \neq \text{null} \land \text{shift}^2(P') \land s(1) = y \land s(0) = e \Rightarrow P \\
s(1) = \text{null} \land \text{shift}^2(P') \land s(1) = y \land s(0) = e \Rightarrow E_{\text{exc}} \\
\text{exc}V = \text{null} \land Q_n[s(0)/\text{result}] \Rightarrow Q_n[s(0)/\text{result}] \\
\text{exc}V \neq \text{null} \land Q_e \Rightarrow E_{l_{\text{exc}}} \\
A \vdash \{ \text{shift}^2(P') \land s(1) = y \land s(0) = e \} \ l_c : \text{callvirt} \ T : m \]

cil invocation rule

The function \( \nabla_B \) (defined in Section 6.2) generates the CIL proof of the routine \( T:m \) from the proof of the routine \( T:m \) in the source. The first implication in the hypothesis shows that the precondition of the instruction \text{callvirt} implies the precondition of the routine \( T:m \), if the target object is not null. The second implication shows that if the target object is null, the precondition of \text{callvirt} implies the precondition at the label \( l_{\text{exc}} \). The third implication shows that the postcondition of \( T:m \) implies the precondition of the successor instruction \( l_d \), if the routine \( m \) does not trigger an exception. Finally, the last implication shows that if the routine \( m \) triggers an exception, the exceptional postcondition \( Q_e \) implies the precondition at the label \( l_{\text{exc}} \).

Translation of the invok-var Rule

The invok-var rule is translated in a similar way to the invocation rule. Let \( T_{tm} \) be the following proof tree:

\[
T_{tm} \equiv \frac{T_{\text{tree}_1}}{A \vdash \{ P \} \ x := y. T:m(e) \ \{ Q_n, Q_e \}}
\]

In the translation:

\[
\nabla_S \left( A \vdash \{ P[w/Z] \} \ x := y. T:m(e) \ \{ Q_n[w/Z], Q_e[w/Z] \}, l_{\text{start}}, l_{\text{next}}, l_{\text{exc}} \right)
\]

we first apply the translation function \( \nabla_S \) to the hypothesis \( T_{tm} \). Finally, we apply the replacement \([w/Z]\) to every bytecode specification generated by the translation of \( T_{tm} \).

6.4.9 Local Rule

The local rule initializes the local variables \( v_1, \ldots, v_n \) with the default values. To translate this rule, we first obtain the translation of the initialization using the translation function
The variables are initialized using the assignment \( v_i := \text{init}(T_i) \) for all variables \( v_1, \ldots, v_n \). After that, the translation of the instruction \( s \) is added using \( \nabla_s \). The translation is defined as follows:

\[
\begin{align*}
\nabla_s (\begin{array}{c}
\text{Tree}_1 \\
\mathcal{A} \vdash \{ P \wedge v_1 = \text{init}(T_1) \} \\
\mathcal{A} \vdash \{ \ldots \wedge v_n = \text{init}(T_n) \} \\
\mathcal{A} \vdash \{ P \} \quad \text{local} \quad T_1 \; v_1; \; \ldots \; T_n \; v_n; \\
\{ s \} \{ Q_n \; Q_e \} \\
\{ Q_n \; Q_e \}
\end{array}) = \\
\n\begin{array}{c}
\text{Tree}_1 \\
\mathcal{A} \vdash \{ P \wedge v_1 = \text{init}(T_1) \} \\
\mathcal{A} \vdash \{ \ldots \wedge v_n = \text{init}(T_n) \} \\
\mathcal{A} \vdash \{ P \wedge v_1 = \text{init}(T_1) \; \wedge \ldots \wedge v_n = \text{init}(T_n) \} \\
\{ s \} \{ Q_n \; Q_e \} \\
\{ Q_n \; Q_e \}
\end{array}
\end{align*}
\]

The bytecode for \( v_1 := \text{init}(T_1) \) establishes \( P \wedge v_1 = \text{init}(T_1) \), which is the precondition of the first instruction of the bytecode for \( v_2 := \text{init}(T_2) \). Applying the same
6.5 Poof Translation of Language-Independent Rules

reasoning, we know that the translation for \( v_i := init(T_i) \) establishes the precondition of its successor. Finally, the bytecode for \( s \) establishes \( Q_n \), which is the postcondition of the rule. Therefore, the translation produces a sequence of valid instruction specifications.

6.4.10 Creation Rule

The translation of creation rule first creates an object of type \( T \). Second, it pops the result in the variable \( x \). Finally, the routine \( \text{make} \) is invoked. To invoke the routine \( \text{make} \), the target object \( x \) and the argument \( e \) are pushed on top of the stack. Then, the routine is invoked using the instruction \( \text{callvirt} \). The proof of the \( \text{callvirt} \) is obtained in a similar way as in the translation of the invocation rule (Section 6.4.8). The translation is defined as follows:

\[
\begin{align*}
\nabla_S & \left( \begin{array}{c}
\{ P \} \quad T@\text{make}(p) \quad \{ Q_n, Q_e \} \\
A & \left( \begin{array}{c}
\{ P \} \quad \left[ \begin{array}{c}
\text{new}($, T)/\text{Current}, \\
$T$/\$, \\
e/p
\end{array} \right] \\
x := \text{create} \{ T \}. \text{make}(e) \\
\{ Q_n[x/\text{Current}], Q_e \}
\end{array} \right) \\
\n\end{array} \right), \quad l_{\text{start}}, l_{\text{next}}, l_{\text{exc}}
\end{align*}
\]

\[
\begin{align*}
\{ P[\text{new}($, T)/\text{Current}, $T$/\$, e/p] \} & \quad l_{\text{start}} : \text{newobj} \ T \\
\{(\text{shift}(P))[s(0)/\text{Current}, e/p] \} & \quad l_b : \text{stloc} \ x \\
\{x \neq \text{Void} \land P[x/\text{Current}, e/p] \} & \quad l_c : \text{ldloc} \ x \\
\nabla_E ( \{ x \neq \text{Void} \land \text{shift}(P[x/\text{Current}, e/p]) \land s(0) = x \}, e, \\
\{ x \neq \text{Void} \land \text{shift}^2(P[x/\text{Current}, e/p]) \} \land s(1) = x \land s(0) = e, \quad l_d \\
\{ x \neq \text{Void} \land \text{shift}^2(P[x/\text{Current}, e/p]) \} \land s(1) = x \land s(0) = e \} & \quad l_c : \text{callvirt} \ T@\text{make}
\end{align*}
\]

6.5 Poof Translation of Language-Independent Rules

In Section 6.2.4 we have presented the translation of language-independent rules for virtual routines and routine implementations. That translation produces a derivation in
the bytecode logic. When the language-independent rules are applied to instructions, the translation has to produce bytecode specifications. For this reason, we present the translation in two parts. In this section, we present the translation of language-independent rules that are applied to instructions.

### 6.5.1 Strength Rule

To translate the strength rule for instructions, the instruction nop instruction is used with precondition $P'$, and then we translate $s_1$ using $\nabla_S$. We assume that the implication $P' \Rightarrow P$ (in the source language) is proven in the same theorem prover used to check the bytecode proof. Since the expressions $P$ and $P'$ are not modified by the translation, the same proof can be used in the bytecode proof to check the implication $P' \Rightarrow P$. The definition is:

$$
\nabla_S \left( \begin{array}{c}
\text{Tree}_1 \\
\mathcal{A} \vdash \{ P \} \ s_1 \ \{ Q_n, Q_e \} \\
\mathcal{A} \vdash \{ P' \} \ s_1 \ \{ Q_n, Q_e \} \\
\mathcal{A} \vdash \{ P' \} \ s_1 \ \{ Q_n, Q_e \} \\
\mathcal{A} \vdash \{ P' \} \ s_1 \ \{ Q_n, Q_e \} \\
\end{array} \right), \ (l_{\text{start}}, l_{\text{next}}, l_{\text{exc}}) =
$$

$$
\{ P' \} , \ l_{\text{start}}: \text{nop}
$$

The instruction nop at label $l_{\text{start}}$ preserves $P'$. Its successor instruction establishes $P$. Therefore, the translation is valid since we know $P' \Rightarrow P$.

### 6.5.2 Weak Rule

The weak rule is translated to CIL in a similar way to the strength rule. First, the instruction $s_1$ is translated using $\nabla_S$. Then, we add a nop instruction with precondition $Q_n$. The proofs of the implications $Q_n \Rightarrow Q'_n$ and $Q_e \Rightarrow Q'_e$ in the source are used in the bytecode proof since the expressions $Q_n, Q'_n, Q_e, Q'_e$ are not modified by the translation. The translation is defined as follows:

$$
\nabla_S \left( \begin{array}{c}
\text{Tree}_1 \\
\mathcal{A} \vdash \{ P \} \ s_1 \ \{ Q_n, Q_e \} \\
\mathcal{A} \vdash \{ P' \} \ s_1 \ \{ Q'_n, Q'_e \} \\
\mathcal{A} \vdash \{ P' \} \ s_1 \ \{ Q'_n, Q'_e \} \\
\mathcal{A} \vdash \{ P' \} \ s_1 \ \{ Q'_n, Q'_e \} \\
\end{array} \right), \ (l_{\text{start}}, l_{\text{next}}, l_{\text{exc}}) =
$$
6.5 Poof Translation of Language-Independent Rules

\[ \nabla_S \left( \frac{Tree_1}{A \vdash \{ P \} \ s_1 \ \{ Q_n \ , \ Q_e \} , \ l_{\text{start}}, l_b, l_{\text{exc}}} \right) \]

\[ \{ Q_n \} \ l_b : \text{nop} \]

6.5.3 Conjunction/Disjunction Rules

The conjunction and disjunction rules are translated to CIL in a similar way. Here, we present the translation for the conjunction rule, the translation of the disjunction rule is analogous. Let \( T_a \) and \( T_b \) be proof trees defined as follows:

\[
T_a \equiv \frac{Tree_1}{A \vdash \{ P' \} \ s_1 \ \{ Q_n' \ , \ Q_e' \} } \quad \quad T_b \equiv \frac{Tree_2}{A \vdash \{ P'' \} \ s_1 \ \{ Q_n'' \ , \ Q_e'' \} } 
\]

To translate the conjunction rule:

\[
\nabla_S \left( \frac{T_a \ T_b}{\{ P' \land P'' \} \ s_1 \ \{ Q_n' \land Q_n'' \ , \ Q_e' \land Q_e'' \} , \ l_{\text{start}}, l_{\text{next}}, l_{\text{exc}}} \right) = \]

we first create the CIL proofs for the two hypotheses:

\[
\nabla_S ( \ T_a , \ l_{\text{start}}, l_{\text{next}}, l_{\text{exc}} )
\]

and

\[
\nabla_S ( \ T_b , \ l_{\text{start}}, l_{\text{next}}, l_{\text{exc}} )
\]

Since the translation is applied to the same instruction \( s_1 \), the embedded instructions in the CIL proofs are by construction the same. With these two proofs, we assemble a third proof by merging, for all instructions except \text{callvirt}, their bytecode specification:

\[ \{ A_i \} \ l_i : \text{instr}_i \]

and

\[ \{ B_i \} \ l_i : \text{instr}_i \]

we obtain
\{A_i \land B_i\} \ l_i : instr_i

If the generated instruction is callvirt, the proof trees have the following form:

\[
\begin{array}{c}
\text{Tree}_1 \\
A \vdash \{ P' \} \ T: m \ \{ Q'_n, Q'_e \} \\
A_i \Rightarrow s(1) \neq \text{null} \land P'[s(1)/\text{this}, s(0)/p][\text{shift}(w')/Z'] \\
Q'_n[s(0)/\text{result}][w'/Z'] \Rightarrow E_{i+1} \\
\end{array}
\]

cil invocation rule

\[
\begin{array}{c}
\text{Tree}_2 \\
A \vdash \{ P'' \} \ T: m \ \{ Q''_n, Q''_e \} \\
B_i \Rightarrow s(1) \neq \text{null} \land P''[s(1)/\text{this}, s(0)/p][\text{shift}(w'')/Z''] \\
Q''_n[s(0)/\text{result}][w''/Z''] \Rightarrow E_{i+1} \\
\end{array}
\]

cil invocation rule

where we assume \(w' \neq w''\) and \(Z' \neq Z''\), and the logical variables used in the proofs are disjoint.

Using these generated proofs, we obtain the conjunction \(A \vdash \{ A_i \land B_i\} \ l_i : \text{callvirt} T: m\) applying the CIL conjunction rule as follows:

\[
\begin{array}{c}
\text{Tree}_1 \quad \text{Tree}_2 \\
A \vdash \{ P' \} \ T: m \ \{ Q'_n, Q'_e \} \quad A \vdash \{ P'' \} \ T: m \ \{ Q''_n, Q''_e \} \\
A \vdash \{ P' \land P'' \} \ T: m \ \{ Q'_n \land Q''_n, Q'_e \land Q''_e \} \\
(A_i \land B_i) \Rightarrow s(1) \neq \text{null} \land (P' \land P'')[s(1)/\text{this}, s(0)/p][\text{shift}(w')/Z'][\text{shift}(w'')/Z''] \\
(Q'_n \land Q''_n)[s(0)/\text{result}][w'/Z'][w''/Z''] \Rightarrow E_{i+1} \\
\end{array}
\]

invok

The translation produces a valid bytecode proof because \(w' \neq w''\), and \(Z' \neq Z''\), and the precondition \(A_i\) does not contain \(Z''\), and \(B_i\) does not contain \(Z'\).

6.5.4 Invariant Rule

To translate the invariant rule, we first generate the translation of the instruction \(s_1\). Then, we add a conjunct \(W\) to every specification produced by that translation. In the
case that the instruction is a callvirt instruction, we apply the same translation as the conjunction rule translation (presented on page 129). The rule is translated as follows:

$$\nabla_S \left( \frac{\text{Tree}_1}{A \vdash \{ P \} \ s_1 \ \{ Q_n, Q_e \}, \ l_{start}, l_{next}, l_{exc}} \right) =$$

$$\nabla_S \left( \frac{\text{Tree}_1}{A \vdash \{ P \ W \} \ s_1 \ \{ Q_n \ W, Q_e \ W \}, \ l_{start}, l_{next}, l_{exc}} \right)$$

where the conjunct $W$ is added to every specification produced by the translation of $s_1$.

### 6.5.5 Substitution Rule

Similar to the invariant rule, the translation of the substitution rule first translates the instruction $s_1$, and then we replace $Z$ by $t$ in each bytecode specification generated by $s_1$. The instruction callvirt instruction is translated applying a similar translation as the conjunction rule translation (presented on page 129).

The definition is:

$$\nabla_S \left( \frac{\text{Tree}_1}{A \vdash \{ P \} \ s_1 \ \{ Q_n, Q_e \}, \ l_{start}, l_{next}, l_{exc}} \right) =$$

$$\nabla_S \left( \frac{\text{Tree}_1}{A \vdash \{ P[t/Z] \} \ s_1 \ \{ Q_n[t/Z], Q_e[t/Z] \}, \ l_{start}, l_{next}, l_{exc}} \right)$$

where the replacement $[t/Z]$ is applied to every bytecode specification generated by $s_1$.

### 6.5.6 All rule and Ex rule

To be able to translate the all-rule and the ex-rule, we assume that the proofs are in a normal form. We assume that the all-rule and ex-rule are applied at the level of routine specifications, so they are not used in the instruction specifications. Thus, these rules are translated in Section 6.2.4.
6.6 Applications

This section illustrates the application of the proof-transforming compiler for the Mate language. In Section 3.1.5 (Chapter 3) we have presented the proof of the function `sum` (this function returns the sum from 1 to `n`). The PTC takes the derivation in the source logic, and produces a derivation in the bytecode logic. In this example, the first rule applied to prove the `sum` function is the class rule. The proof-transforming compiler produces the following application of the CIL class rule:

\[
\begin{array}{l}
\{ \tau(\text{Current}) < \text{MATH} \land n > 1 \} \quad \text{MATH:sum} \quad \{ \text{Result} = (n * (n + 1))/2, \text{false} \} \\
\{ \tau(\text{Current}) = \text{MATH} \land n > 1 \} \quad \text{impl}(\text{MATH}, \text{sum}) \quad \{ \text{Result} = (n * (n + 1))/2, \text{false} \} \\
\{ \tau(\text{Current}) \preceq \text{MATH} \land n > 1 \} \quad \text{MATH:sum} \quad \{ \text{Result} = (n * (n + 1))/2, \text{false} \}
\end{array}
\]

In the hypothesis, the proof for the virtual routine `MATH:sum` is obtained applying the translation function $\nabla_B$ to the hypothesis of the source rule. This application returns the following application of the strength rule:

\[
\begin{array}{l}
\{ \text{false} \} \quad \text{MATH:sum} \quad \{ \text{Result} = (n * (n + 1))/2, \text{false} \} \\
\{ \tau(\text{Current}) < \text{MATH} \land n > 1 \} \quad \text{MATH:sum} \quad \{ \text{Result} = (n * (n + 1))/2, \text{false} \}
\end{array}
\]

The proof for the second hypothesis is obtained by the translation function $\nabla_B$. In this case, the translation of the routine implementation rule is applied. The proof of the body of the routine `sum` is obtained by the translation function $\nabla_S$. This proof is presented in Figure 6.1

To translate the body of the routine `sum`, the proof-transforming compiler takes the proof of Figure 3.5 (presented on page 32), and produces as result the bytecode proof in Figure 6.1. 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 body of the loop. Finally, the until expression is translated in lines IL020-IL025.

To show that the bytecode proof is a valid proof, one has to show that each precondition implies the weakest precondition of the successor instruction. In most of the cases, this proof is done by applying the definition of weakest precondition (a detailed example is shown in Section 5.2.4 page 103). The most interesting cases of this proof are lines IL009-IL010, and lines IL011-IL012 (the translation of the weak and strength rule). These two implications are proven since the proofs have been developed for the source
6.6 Applications

{ $n > 1$ }
{ $n > 1 \land s(0) = 0$ }
{ $n > 1 \land Result = 0$ }
{ $n > 1 \land Result = 0 \land s(0) = 0$ }

// from body
{ $n > 1 \land Result = 0 \land i = 0$ }
{ $n > 1 \land Result = 0 \land i = 0 \land s(0) = 1$ }
{ $n > 1 \land Result = 1 \land i = 0$ }
{ $n > 1 \land Result = 1 \land s(0) = 2$ }

// weak and strength rules
{ $n > 1 \land Result = 1 \land i = 2$ }
{ $i \neq n + 1 \land Result = ((i - 1) * i)/2 \land n + 1 \geq i \land i > 1$ }
{ $i \neq n + 1 \land Result = ((i - 1) * i)/2 \land n + 1 \geq i \land i > 1$ }

// loop body
{ $i \neq n + 1 \land Result + i = (i * (i + 1))/2 \land n + 1 \geq i \land i > 1$ }
{ $i \neq n + 1 \land Result + i = (i * (i + 1))/2 \land n + 1 \geq i \land i > 1 \land s(0) = Result$ }
{ $i \neq n + 1 \land Result + i = (i * (i + 1))/2 \land n + 1 \geq i \land i > 1 \land s(1) = Result \land s(0) = i$ }
{ $i \neq n + 1 \land Result + i = (i * (i + 1))/2 \land n + 1 \geq i \land i > 1 \land s(0) = Result + i$ }
{ $i \neq n + 1 \land Result = (i * (i + 1))/2 \land n + 1 \geq i \land i > 1$ }
{ $i \neq n + 1 \land Result = (i * (i + 1))/2 \land n + 1 \geq i \land i > 1 \land s(0) = i$ }
{ $i \neq n + 1 \land Result = (i * (i + 1))/2 \land n + 1 \geq i \land i > 1 \land s(1) = i \land s(0) = 1$ }
{ $i \neq n + 1 \land Result = (i * (i + 1))/2 \land n + 1 \geq i \land i > 1 \land s(0) = i + 1$ }

// until expression
{ $Result = ((i - 1) * i)/2 \land n + 1 \geq i \land i > 1$ }
{ $Result = ((i - 1) * i)/2 \land n + 1 \geq i \land i > 1 \land s(0) = i$ }
{ $Result = ((i - 1) * i)/2 \land n + 1 \geq i \land i > 1 \land s(1) = i \land s(0) = n$ }
{ $Result = ((i - 1) * i)/2 \land n + 1 \geq i \land i > 1 \land s(2) = i \land s(1) = n \land s(0) = 1$ }
{ $Result = ((i - 1) * i)/2 \land n + 1 \geq i \land i > 1 \land s(1) = i \land s(0) = n + 1$ }
{ $Result = ((i - 1) * i)/2 \land n + 1 \geq i \land i > 1 \land s(0) = (i = n + 1)$ }

IL001 : ldc 0
IL002 : stloc Result
IL003 : ldc 0
IL004 : stloc i
IL005 : ldc 1
IL006 : stloc Result
IL007 : ldc 2
IL008 : stloc i
IL009 : nop
IL010 : br IL020
IL011 : nop
IL012 : ldloc Result
IL013 : ldloc i
IL014 : add
IL015 : stloc Result
IL016 : ldloc i
IL017 : ldc 1
IL018 : add
IL019 : stloc i
IL020 : ldc i
IL021 : ldloc n
IL022 : ldc 1
IL023 : add
IL024 : ceq
IL025 : bfalse IL011

Fig. 6.1: Bytecode Proof generated by the PTC from the Source Proof of Figure 3.5
program, and these proofs are translated to CIL. Thus, the translation introduces two lemmas that show the precondition at IL009 implies the precondition at IL010, and that the precondition at IL011 implies the precondition at IL012. The proofs of these lemmas are identical to the proofs in the source.

6.7 Soundness Theorems

To be able to execute mobile code in a safe way, a soundness proof is required only for components of the trusted code base. Although PTCs are not part of the trusted code base, from the point of view of the code producer, the PTC should always generate valid proofs to avoid that the produced bytecode is rejected by the proof checker.

It is thus desirable to prove the soundness of the translation. Soundness informally means that the translation produces valid bytecode proofs. It is not enough, however, to produce a valid proof, because the compiler could generate bytecode proofs where every precondition is false. The theorem states that if we have a valid source proof for the routine $m$, then (1) the generated bytecode proof tree is valid, and (2) the evaluation of the pre and postcondition of the routine $m$ is equal to the evaluation of the translation to CIL of the pre and postcondition of $m$ respectively. We split the theorem into two theorems, one expresses soundness of the proof translation, and the other one expresses soundness of the contract translation:

**Theorem 3** (Soundness of Routine Translation).

\[
A \vdash \{ P \} \quad m \quad \{ Q_n, Q_e \} \quad \text{then} \quad \nabla_B \left( \frac{\text{Tree}_1}{A \vdash \{ P \} \quad m \quad \{ Q_n, Q_e \}} \right)
\]

To prove Theorem 3, we define a soundness theorem of the proof transformation for instructions.

The theorem for instruction translation states that if (1) we have a valid source proof for the instruction $s_1$, and (2) we have a proof translation from the source proof that produces the instructions $I_{l_{\text{start}}}...I_{l_{\text{end}}}$, and their respective preconditions $E_{l_{\text{start}}}...E_{l_{\text{end}}}$, and (3) the normal postcondition in the source logic implies the next precondition of the last generated instruction (if the last generated instruction is the last instruction of the method, we use the normal postcondition in the source logic), and (4) the exceptional postcondition in the source logic implies the precondition at the target label $l_{\text{exc}}$ but
6.7 Soundness Theorems

considering the value stored in the stack of the bytecode, then every bytecode specification
holds ($\vdash \{E_i\} I_i$). The theorem is the following:

**Theorem 4** (Soundness of Instruction Translator).

\[
\begin{align*}
\vdash & \quad \text{Tree}_1 \\
\mathcal{A} \vdash \left\{ P \right\} & s_1 \left\{ Q_n, Q_e \right\} \quad \land \\
(I_{start} \ldots I_{end}) &= \nabla_s \left( \mathcal{A} \vdash \left\{ P \right\} s_1 \left\{ Q_n, Q_e \right\}, l_{start}, l_{end+1}, l_{exc} \right) \quad \land \\
(Q_n &\Rightarrow E_{l_{end+1}}) \quad \land \\
(Q_e \land \text{exc} \neq \text{null} \land s(0) = \text{exc}V) &\Rightarrow E_{l_{exc}} \quad \land \\
\forall l &\in l_{start} \ldots l_{end} : \vdash \{E_i\} I_l
\end{align*}
\]

The full proofs can be found in Appendix C. The proof runs by induction on the
structure of the derivation tree for $\{P\} s_1 \{Q_n, Q_e\}$. 

Chapter 7

Eiffel-Specific Aspects of Proof-Transforming Compilation

In the previous chapter, we have presented a proof-transforming compiler for a core object-oriented language. The translation is fairly straightforward because the source and the target language are very similar. This is also the case of other PTCs that have been developed from Java to bytecode [6, 15, 103]. But the difficulty of the problem grows with the conceptual distance between the semantic models of the source and target languages. In this chapter, the source language is Eiffel, whose object model and type system differ significantly from the assumptions behind CIL, the target language. In particular, Eiffel supports multiple inheritance and a specific form of exception handling. This has required, in the implementation of Eiffel for .NET (which goes through CIL code), the design of original compilation techniques. In particular [74], the compilation of each Eiffel class produces two CIL types: an interface, and an implementation class which implements it. If either the source proof or the source specification expresses properties about the type structure of the Eiffel program, the same property has to be generated for the bytecode.

The translation of these properties raises challenges illustrated by the following example (interface only, implementation omitted) involving a reflective capability: the feature `type`, which gives the type of an object.

```plaintext
merge (other: LINKED_LIST [G]): LINKED_LIST [G]
   - - Merge other into current structure returning a new LINKED_LIST
   require
      is_linked_list : other.type.conforms_to (LINKED_LIST [G].type)
      same_type: Current.type.is_equal(other.type)
   ensure
      result_type : Result.type.is_equal(LINKED_LIST [G].type)
```
The function `merge` is defined in the class `LINKED_LIST`. The precondition of `merge` expresses that the type of `other` is a subtype of `LINKED_LIST` and the types of `Current` and `other` are equal. The first precondition is for illustration propose. The postcondition expresses that the type of `Result` is equal to `LINKED_LIST`.

The compilation of the class `LINKED_LIST` produces the CIL interface `LINKED_LIST_INTERF` and the implementation class `LINKED_LIST_IMP`. A correct PTC has to map the type `LINKED_LIST` in the clause `is_linked_list` (line 4) to the CIL interface `LINKED_LIST_INTERF` because in the target model decedents of the Eiffel class `LINKED_LIST` inherit from the interface `LINKED_LIST_INTERF` in CIL and not from `LINKED_LIST_IMP`. To translate the postcondition, we use the implementation class `LINKED_LIST_IMP` because this property expresses that the type of `Result` is equal to `LINKED_LIST`. Thus, the PTC has to map Eiffel classes to CIL interfaces or Eiffel classes to CIL classes depending of the function used to express the source property.

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

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

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

This chapter is based on the published works [96, 95].

### 7.1 Contract Translator

The contract translator maps Eiffel contracts into first order logic (FOL). In Chapter 6, the contract translator is straightforward. (Its definition is $\nabla_C(e) = e$). The challenging problem in the contract translator in this chapter is produced by the impedance mismatch between Eiffel and CIL. To be able to translate contracts, we use deeply-embedded Eiffel expressions.

#### 7.1.1 Translation Basics

In Hoare triples, pre- and postconditions may refer to the structure of the Eiffel program. Therefore, in our logic, pre- and postconditions are deeply-embedded Eiffel expressions,
extended with universal and existential quantifiers. The proof translation proceeds in two
steps: first, translation of pre-and postconditions into FOL using the translation function
presented in this section; then, translation of the proof using the functions presented
in Section 7.2. We have defined a deep embedding of the Eiffel expressions used in the
contract language. Then, we have defined translation functions to FOL. The datatype
definitions, the translation functions and their soundness proof are formalized in Isabelle
(this formalization can be found in our technical report [95]).

7.1.2 Datatype Definitions

Eiffel contracts are based on boolean expressions, extended (for postconditions) with
the old notation. They can be constructed using the logical operator ¬, ∧, ∨, AndThen,
OrElse, Xor, implies, expressions equality <, >, ≤, ≥, and the type functions ConformsTo,
IsEqual, and IsNotEqual. Expressions are constants, local variables and arguments, at-
tributes, routine calls, precursor expressions, creation expressions, arithmetic expressions,
old expressions, boolean expressions, and Void. Arguments are treated as local variables
using the sort RefVar to minimize the datatype definition. Furthermore, boolean variables
are not introduced in the definition BoolExp. They are treated as local variables using the
sort RefVar. We assume routines have exactly one argument.

```plaintext
datatype EiffelContract = Require BoolExpr
                      | Ensure BoolExpr

datatype BoolExpr = Const Bool
                  | Neg BoolExpr
                  | And BoolExpr BoolExpr
                  | Or BoolExpr BoolExpr
                  | AndThen BoolExpr BoolExpr
                  | OrElse BoolExpr BoolExpr
                  | Xor BoolExpr BoolExpr
                  | Impl BoolExpr BoolExpr
                  | Eq Expr Expr
                  | NotEq Expr Expr
                  | Less Expr Expr
                  | Greater Expr Expr
                  | LessE Expr Expr
                  | GreaterE Expr Expr
                  | Type TypeFunc

datatype TypeFunc = ConformsTo TypeExpr TypeExpr
                   | IsEqual TypeExpr TypeExpr
                   | IsNotEqual TypeExpr TypeExpr
```
datatype TypeExpr = EType EiffelType
| Type Expr

datatype Expr = ConstInt Int
| RefVar VarId
| Att ObjID Attrib
| CallR CallRoutine
| Precursor EiffelType Routine Argument
| Create EiffelType Routine Argument
| Plus Expr Expr
| Minus Expr Expr
| Mul Expr Expr
| Div Expr Expr
| Old Expr
| Bool BoolExpr
| Void

datatype CallRoutine = Call Expr Routine Argument

datatype Argument = Argument Expr

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

datatype EiffelType = Boolean
| Integer
| EClass (cID : ClassID)
| None

Variables, attributes and routines are defined as follows:

datatype Var = Var VarID EiffelType
| Result EiffelType
| Current EiffelType

datatype Attrib = Attr (aID : attribID) EiffelType

datatype Routine = Routine RoutineID EiffelType EiffelType

7.1.3 Mapping Eiffel Types to CIL

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

\[ \text{datatype } \text{CilType} = \text{CilBoolean} \mid \text{CilInteger} \mid \text{Interface } \text{classID} \mid \text{CilClass } \text{classID} \mid \text{NullT} \]

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

\[
\begin{align*}
\nabla_{\text{interface}} & : \text{EiffelType} \rightarrow \text{CilType} \\
\nabla_{\text{class}} & : \text{EiffelType} \rightarrow \text{CilType} \\
\nabla_{\text{class}}(\text{Boolean}) & = \text{CilBoolean} \\
\nabla_{\text{class}}(\text{Integer}) & = \text{CilInteger} \\
\nabla_{\text{class}}(\text{EClass } n) & = \text{CilClass } n \\
\nabla_{\text{class}}(\text{None}) & = \text{NullT} \\
\n\end{align*}
\]

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

\[ \text{datatype } \text{CilMethod} = \text{Method } \text{methodID} \text{ CilType CilType} \]

\[
\begin{align*}
\nabla_{\text{R}} & : \text{Routine} \rightarrow \text{CilMethod} \\
\nabla_{\text{R}}(\text{Routine } n \ t_1 \ t_2) & = (\text{Method } n \ (\nabla_{\text{Interface } t_1} \ (\nabla_{\text{Interface } t_2})) \\
\end{align*}
\]

7.1.4 Translation Functions

The translation of the specification relies on five translation functions: (1) \( \nabla_{\text{C}} \) takes a boolean expression and returns a function that takes two stores and a state and returns a value; (2) \( \nabla_{\text{Exp}} \) translates expressions; (3) \( \nabla_{\text{T}} \) translates type functions (conforms to, is equal, and is not equal); (4) \( \nabla_{\text{Call}} \) translates a routine call; and (5) \( \nabla_{\text{Arg}} \) translates arguments. These functions use two object stores, the second one is used to evaluate old expressions. \text{State} is a mapping from variables to values (\( \text{VarId} \rightarrow \text{Value} \)). The signatures of these functions are the following:

\[
\begin{align*}
\nabla_{\text{C}} : & \text{BoolExpr} \rightarrow (\text{ObjectStore } \rightarrow \text{ObjectStore } \rightarrow \text{State } \rightarrow \text{Value}) \\
\nabla_{\text{Exp}} : & \text{Expr} \rightarrow (\text{ObjectStore } \rightarrow \text{ObjectStore } \rightarrow \text{State } \rightarrow \text{Value}) \\
\nabla_{\text{T}} : & \text{TypeFunc} \rightarrow (\text{ObjectStore } \rightarrow \text{ObjectStore } \rightarrow \text{State } \rightarrow \text{Value}) \\
\nabla_{\text{Call}} : & \text{CallRoutine} \rightarrow (\text{ObjectStore } \rightarrow \text{ObjectStore } \rightarrow \text{State } \rightarrow \text{Value}) \\
\nabla_{\text{Arg}} : & \text{Argument} \rightarrow (\text{ObjectStore } \rightarrow \text{ObjectStore } \rightarrow \text{State } \rightarrow \text{Value}) \\
\end{align*}
\]
To clarify the translation of contracts, we repeat the definition of the data type \textit{Value}:

\textbf{data type} \textit{Value} = \texttt{boolV} \ (aB : Bool) \\
\quad \mid \texttt{intV} \ (aI : Int) \\
\quad \mid \texttt{objV} \ \textit{ClassId ObjId} \\
\quad \mid \texttt{voidV}

where the notation \((aB : \texttt{Bool})\) means, given a value \(v\) of type boolean, \(aB(v)\) returns its boolean expression. The definition of the function \(\nabla_C\) is the following:

\[
\begin{align*}
\nabla_C(\text{Const } b) &= \lambda (h_1, h_2 : \text{ObjectStore}) \ (s : \text{State}) : \ (\text{boolV } b) \\
\nabla_C(\text{Neg } b) &= \lambda (h_1, h_2 : \text{ObjectStore}) \ (s : \text{State}) : \\
& \quad \ (\text{boolV } \neg(aB(\nabla_C b \ h_1 \ h_2 s))) \\
\nabla_C(\text{And } b_1 \ b_2) &= \lambda (h_1, h_2 : \text{ObjectStore}) \ (s : \text{State}) : \\
& \quad \ (\text{boolV}(aB(\nabla_C b_1 \ h_1 \ h_2 s)) \land (aB(\nabla_C b_2 \ h_1 \ h_2 s))) \\
\nabla_C(\text{Or } b_1 \ b_2) &= \lambda (h_1, h_2 : \text{ObjectStore}) \ (s : \text{State}) : \\
& \quad \ (\text{boolV}(aB(\nabla_C b_1 \ h_1 \ h_2 s)) \lor (aB(\nabla_C b_2 \ h_1 \ h_2 s))) \\
\nabla_C(\text{AndThen } b_1 \ b_2) &= \lambda (h_1, h_2 : \text{ObjectStore}) \ (s : \text{State}) : \\
& \quad \ (\text{boolV}(aB(\nabla_C b_1 \ h_1 \ h_2 s)) \land (aB(\nabla_C b_2 \ h_1 \ h_2 s))) \\
\nabla_C(\text{OrElse } b_1 \ b_2) &= \lambda (h_1, h_2 : \text{ObjectStore}) \ (s : \text{State}) : \\
& \quad \ (\text{boolV}(aB(\nabla_C b_1 \ h_1 \ h_2 s)) \lor (aB(\nabla_C b_2 \ h_1 \ h_2 s))) \\
\nabla_C(\text{Xor } b_1 \ b_2) &= \lambda (h_1, h_2 : \text{ObjectStore}) \ (s : \text{State}) : \\
& \quad \ (\text{boolV}(aB(\nabla_C b_1 \ h_1 \ h_2 s)) \lor (aB(\nabla_C b_2 \ h_1 \ h_2 s))) \\
\nabla_C(\text{Impl } b_1 \ b_2) &= \lambda (h_1, h_2 : \text{ObjectStore}) \ (s : \text{State}) : \\
& \quad \ (\text{boolV}(aB(\nabla_C b_1 \ h_1 \ h_2 s)) \rightarrow (aB(\nabla_C b_2 \ h_1 \ h_2 s))) \\
\nabla_C(\text{Eq } e_1 \ e_2) &= \lambda (h_1, h_2 : \text{ObjectStore}) \ (s : \text{State}) : \\
& \quad \ (\text{boolV} (aI(\text{Exp } e_1 \ h_1 \ h_2 s) = (aI(\text{Exp } e_2 \ h_1 \ h_2 s))) \\
\nabla_C(\text{NotEq } e_1 \ e_2) &= \lambda (h_1, h_2 : \text{ObjectStore}) \ (s : \text{State}) : \\
& \quad \ (\text{boolV} (aI(\text{Exp } e_1 \ h_1 \ h_2 s) \neq (aI(\text{Exp } e_2 \ h_1 \ h_2 s))) \\
\nabla_C(\text{Less } e_1 \ e_2) &= \lambda (h_1, h_2 : \text{ObjectStore}) \ (s : \text{State}) : \\
& \quad \ (\text{boolV} (aI(\text{Exp } e_1 \ h_1 \ h_2 s) < (aI(\text{Exp } e_2 \ h_1 \ h_2 s))) \\
\nabla_C(\text{Greater } e_1 \ e_2) &= \lambda (h_1, h_2 : \text{ObjectStore}) \ (s : \text{State}) : \\
& \quad \ (\text{boolV} (aI(\text{Exp } e_1 \ h_1 \ h_2 s) > (aI(\text{Exp } e_2 \ h_1 \ h_2 s)))
\end{align*}
\]
7.1 Contract Translator

\[ \nabla_C(\text{LessE } e_1 e_2) = \lambda (h_1, h_2 : \text{ObjectStore}) (s : \text{State}) : \\
\quad (\text{boolV}(aI(\nabla_{\text{Exp}} e_1 h_1 h_2 s)) \leq (aI(\nabla_{\text{Exp}} e_2 h_1 h_2 s))) \]

\[ \nabla_C(\text{GreaterE } e_1 e_2) = \lambda (h_1, h_2 : \text{ObjectStore}) (s : \text{State}) : \\
\quad (\text{boolV}(aI(\nabla_{\text{Exp}} e_1 h_1 h_2 s)) \geq (aI(\nabla_{\text{Exp}} e_2 h_1 h_2 s))) \]

\[ \nabla_C(\text{Type } e) = \lambda (h_1, h_2 : \text{ObjectStore}) (s : \text{State}) : (\nabla_T e h_1 h_2 s) \]

The function \( \nabla_T \) maps the Eiffel types to CIL. The Eiffel function \( \text{ConformsTo} \) is mapped to the function \( \preceq^e \) (subtyping in CIL). Its types are translated to interfaces using the function \( \nabla_{\text{Interface}} \). The function \( \text{IsEqual} \) is translated using the function \( = \) (types equality in CIL), and the function \( \text{IsNotEqual} \) to \( \neq \).

Its types are translated to CIL classes using the function \( \nabla_{\text{Class}} \). The function \( \nabla_t \) is defined as follows:

\[ \nabla_T(\text{ConformsTo } t_1 t_2) = \lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) : \\
\quad (\text{boolV}(\nabla_{\text{Interface}}(\nabla_{\text{Type}} t_1)) \preceq^e (\nabla_{\text{Interface}}(\nabla_{\text{Type}} t_2))) \]

\[ \nabla_T(\text{IsEqual } t_1 t_2) = \lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) : \\
\quad (\text{boolV}(\nabla_{\text{Class}}(\nabla_{\text{Type}} t_1)) = (\nabla_{\text{Class}}(\nabla_{\text{Type}} t_2))) \]

\[ \nabla_T(\text{IsNotEqual } t_1 t_2) = \lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) : \\
\quad (\text{boolV}(\nabla_{\text{Class}}(\nabla_{\text{Type}} t_1)) \neq (\nabla_{\text{Class}}(\nabla_{\text{Type}} t_2))) \]

The function \( \nabla_{\text{type}} \) given a type expression returns its Eiffel type:

\[ \nabla_{\text{type}} : \text{TypeExp} \rightarrow \text{EiffelType} \]

\[ \nabla_{\text{Type}}(\text{EType } t) = t \]

\[ \nabla_{\text{Type}}(\text{Expression } e) = (\text{typeOf } e) \]

The function \( \nabla_{\text{Exp}} \) translates local variables using the state \( s \). Creation instructions are translated using the functions of the object store \( \text{new} \) and \( \ldots(\ldots) \). The translation of old expressions uses the second \( \text{store} \) to map the expression \( e \) to CIL. To facilitate the description of this translation, we first repeat the signatures of the functions that are applied to the object store (for more detail about these functions see Section 3.1.2):

\[ \ldots(\ldots) : \text{ObjectStore} \times \text{Location} \rightarrow \text{Value} \]

\[ \text{alive} : \text{Value} \times \text{ObjectStore} \rightarrow \text{Bool} \]

\[ \text{new} : \text{ObjectStore} \times \text{ClassId} \rightarrow \text{Value} \]

\[ \ldots(\ldots) : \text{ObjectStore} \times \text{Location} \times \text{Value} \rightarrow \text{ObjectStore} \]

\[ \ldots(\ldots) : \text{ObjectStore} \times \text{ClassId} \rightarrow \text{ObjectStore} \]

The definition of \( \nabla_{\text{Exp}} \) is:
\[ \nabla_{\text{Exp}}(\text{ConstInt } i) = \lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) : (\text{intV } i) \]

\[ \nabla_{\text{Exp}}(\text{RefVar } v) = \lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) : (s(v)) \]

\[ \nabla_{\text{Exp}}(\text{Att } ob a) = \lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) : h_1 (\text{Loc (aID } ob)) \]

\[ \nabla_{\text{Exp}}(\text{CallR } crt) = \lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) : (\nabla_{\text{Call}} \text{ crt } h_1 h_2 s) \]

\[ \nabla_{\text{Exp}}(\text{Precursor } t_1 rt \text{ par}) = \lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) : (\text{invokeValCIL } h_1 (\text{\nabla_r } rt) (s (\text{Currentt}_1)) (\nabla_{\text{arg }} \text{ par } h_1 h_2 s)) \]

\[ \nabla_{\text{Exp}}(\text{Create } t rt p) = \lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) : (\text{new } (h_1((\text{cID } t))) (\text{cID } t)) \]

\[ \nabla_{\text{Exp}}(\text{Plus } e_1 e_2) = \lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) : (\text{intV}((\text{aI} (\nabla_{\text{Exp}} e_1 h_1 h_2 s)) + (\text{aI}(\nabla_{\text{Exp}} e_2 h_1 h_2 s)))) \]

\[ \nabla_{\text{Exp}}(\text{Minus } e_1 e_2) = \lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) : (\text{intV}((\text{aI}(\nabla_{\text{Exp}} e_1 h_1 h_2 s)) - (\text{aI}(\nabla_{\text{Exp}} e_2 h_1 h_2 s)))) \]

\[ \nabla_{\text{Exp}}(\text{Mul } e_1 e_2) = \lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) : (\text{intV}((\text{aI}(\nabla_{\text{Exp}} e_1 h_1 h_2 s)) * (\text{aI}(\nabla_{\text{Exp}} e_2 h_1 h_2 s)))) \]

\[ \nabla_{\text{Exp}}(\text{Div } e_1 e_2) = \lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) : (\text{intV}((\text{aI}(\nabla_{\text{Exp}} e_1 h_1 h_2 s))/ (\text{aI}(\nabla_{\text{Exp}} e_2 h_1 h_2 s)))) \]

\[ \nabla_{\text{Exp}}(\text{Old } e) = \lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) : (\nabla_{\text{Exp}} e h_2 h_2 s) \]

\[ \nabla_{\text{Exp}}(\text{Void }) = \lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) : (\text{VoidV}) \]

The function \( \nabla_{\text{Call}} \) is defined as follows:

\( \nabla_{\text{Call}}(\text{Call } e_1 rt p) = \lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) : (\text{CilInvokeVal } h_1 (\text{\nabla_r } rt) (\text{\nabla_{\text{Exp}} } e_1 h_1 h_2 s)(\nabla_{\text{Arg }} p h_1 h_2 s)) \)

The function \( \text{CilInvokeVal} \) takes a CIL method \( m \) and two values (its argument \( p \) and invoker \( e_1 \)) and returns the value of the result of invoking the method \( m \) with the invoker \( e_1 \) and argument \( p \).

The definition of the function \( \nabla_{\text{Arg}} \) is the following:

\( \nabla_{\text{Arg}}(\text{Argument } e) = \lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) : (\text{\nabla_{\text{Exp}} } e h_1 h_2 s) \)

### 7.1.5 Example Translation

To be able to translate contracts, first we embed the contracts in Isabelle using the above data type definitions. Then, we apply the translation function \( \nabla_C \) which produces the
7.2 Proof Translator

contracts in FOL. Following, we present the embedding of the contracts of the function \textit{merge} presented in the introduction of this chapter. Its precondition is embedded as follows:

$$\text{Type ( \textbf{ConformsTo (Type (\textit{RefVar other}) \}) \ (EType \ \textit{LINKED\_LIST}[G]) \} )}$$

$$\text{Type ( \textbf{isEqual (Type (\textit{RefVar Current}) \}) \ (Type (\textit{RefVar other}) \}) )}$$

The deep embedding of \textit{merge}'s postcondition is as follows:

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

The application of the function \(\nabla_C\) to the precondition produces the following expression:

$$\lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) :$$
$$\begin{align*}
&\text{boolV(typeOf other) } \leq_c \text{(interface } \textit{LINKED\_LIST}[G]) \\
&\lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) : \text{boolV(typeOf Current) } = \text{(typeOf other)}
\end{align*}$$

The result of the application of the function \(\nabla_C\) to the deep embedding of \textit{merge}'s postcondition is the following:

$$\lambda (h_1, h_2 : \text{ObjectStore})(s : \text{State}) :$$
$$\begin{align*}
&\text{boolV(typeOf Current) } = \text{(CilClass } \textit{LINKED\_LIST}[G])
\end{align*}$$

In the precondition, the type \(\textit{LINKED\_LIST}[G]\) is translated to the interface \(\textit{LINKED\_LIST}[G]\) because the precondition uses the function \textbf{ConformsTo}. However, in the postcondition, the type \(\textit{LINKED\_LIST}[G]\) is translated to the class \(\textit{LINKED\_LIST}[G]\) because it uses the function \textbf{isEqual}. The PTC can translate these types because it takes deeply-embedded Eiffel expressions as input.

7.2 Proof Translator

In Section 6.1 and Section 6.2, we have presented the proof-transforming compilation of virtual routines and routines implementations, and the translation of language-independent rules. The Eiffel proof-transforming compiler uses the same functions to translate routines specifications. This PTC extends the definitions of the transformation functions \(\nabla_S\) and \(\nabla_E\) to handle the Eiffel expressions \textbf{and then} \textbf{or else}, and \textbf{rescue} clauses and once routines. These functions yield a sequence of Bytecode instructions and their specifications.
7.2.1 Transformation Function Basics

The function $\nabla_E$ generates a bytecode proof from a source expression and a precondition for its evaluation. The function $\nabla_S$ generates a bytecode proof from a source proof. These functions are defined as a composition of the translations of the proof’s sub-trees. They have the signatures:

$$\nabla_E : \text{Precondition} \times \text{Expression} \times \text{Postcondition} \times \text{Label} \rightarrow \text{BytecodeProof}$$

$$\nabla_S : \text{ProofTree} \times \text{Label} \times \text{Label} \times \text{Label} \rightarrow \text{BytecodeProof}$$

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

The proof translation will now be presented for the Eiffel expressions and then and or else, and the rescue clauses and once routines. The definition of $\nabla_E$ is presented in Section 7.2.2. The function $\nabla_S$ is presented in Section 7.2.3.

7.2.2 Proof Translation of Eiffel Expressions

In Chapter 6, Section 6.3, we have presented a translation for expressions. The translation includes simple expressions such unary and binary expressions. Eiffel introduces the expressions and then, and or else. The semantics of $e_1 \text{ and then } e_2$ is as follows: if the evaluation of $e_1$ yields false, then the evaluation of $e_1 \text{ and then } e_2$ yields false without evaluating $e_2$; otherwise the evaluation of $e_1 \text{ and then } e_2$ yields the result of the evaluation of $e_2$. The semantics of $e_1 \text{ or else } e_2$ is: if the evaluation of $e_1$ yields true, then this expression yields true without evaluating $e_2$; otherwise it yields the evaluation of $e_2$.

The CIL language does not contain any instructions to evaluate the expressions and then, and or else. Thus, the translation for expressions needs to be extended.

Expressions and then

To translate the expression and then, first, the expression $e_1$ is pushed on top of the stack. If $e_1$ evaluates to false (label $l_b$), control is transferred to $l_c$, and the constant false is pushed on top of the stack because $e_1 \text{ and then } e_2$ evaluates to false. If $e_1$ evaluates to true, then the expression $e_2$ is pushed on top of the stack (at label $l_b$), and control is transfer to the next instruction. Thus, the evaluation $e_1 \text{ and then } e_2$ returns $e_2$ because $e_1$ evaluates to true. To be able to translate and then expressions, we assume the expression $e_1 \land e_2$ is evaluated left to right; if $e_1$ evaluates to false, then $e_2$ is not evaluated. The translation is defined as follows:
\[ \nabla_E(Q \land \text{unshift}(P[e_1 \land e_2/s(0)]), \ e_1 \text{ and then } e_2, \ shift(Q) \land P, l_a, l_{end}) = \]

\[
\nabla_E(Q \land \text{unshift}(P[e_1 \land e_2/s(0)]), \ e_1, \ shift(Q) \land P[s(0) \land e_2/s(0)], l_a, l_{end})
\{\text{shift}(Q) \land P[s(0) \land e_2/s(0)]\} \ l_b : \text{brfalse} \ l_c
\]

\[
\nabla_E(Q \land \text{unshift}(P[(true \land e_2)/s(0)]), \ e_2, \ shift(Q) \land P[(true \land s(0)/s(0)], l_c, l_{end})
\{\text{shift}(Q) \land P[true \land s(0)/s(0)]\} \ l_d : \text{br} \ l_{end}
\]

\[
\{Q \land \text{unshift}(P[false \land e_2/s(0)])\} \ l_e : \text{ldc} \ false
\]

The translation for \(e_1\) establishes \(shift(Q) \land P[s(0) \land e_2/s(0)]\) which is the precondition of the instruction at \(l_b\). The instruction \text{brfalse}, has two possible successors: \(l_c\) and \(l_e\). If the top of the stack is \(true\) then applying the weakest precondition one proves

\[Q \land \text{unshift}(P[(true \land e_2)/s(0)]\]

holds. If the top of the stack is \(false\), we can prove

\[shift(Q) \land P[true \land s(0)/s(0)]\]

applying the definition of \text{wp}. Finally, the translation for \(e_2\) establishes

\[shift(Q) \land P[(true \land s(0)/s(0)]\]

which is the precondition of its successor. Therefore, the translation is valid.

**Expressions or else**

The expressions \text{or else} are translated in a similar way than \text{and then} expressions. At label \(l_a\), the expression \(e_1\) is translated using the function \(\nabla_E\). This translation pushes the evaluation of \(e_1\) on top of the stack. If \(e_1\) is \(true\), control is transferred to \(l_c\) where the constant \(true\) is pushed on top the stack. If \(e_1\) evaluates to \(false\), the expression \(e_2\) is translated (at label \(l_c\)), and control is transferred to the next instruction. Similar to \text{and then} expressions, we assume the expression \(e_1 \lor e_2\) is evaluated left to right, if \(e_1\) evaluates to \(true\) then \(e_2\) is not evaluated. The definition is:
\[ \nabla_E(Q \land \text{unshift}(P[e_1 \text{ or else } e_2]s(0)), e_1 \text{ or else } e_2, \text{shift}(Q) \land P, l_a, l_{end}) = \]

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

\{ \text{shift}(Q) \land P[s(0) \lor e_2]s(0) \} l_b : \text{brtrue } l_c

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

\{ \text{shift}(Q) \land P[\text{false} \lor s(0)]s(0) \} l_d : \text{br } l_{end}

\{ Q \land \text{unshift}(P[\text{true} \lor e_2]s(0)) \} l_e : \text{ldec } true

The translation for \( e_1 \) establishes \( \text{shift}(Q) \land P[s(0) \lor e_2]s(0) \) which is the precondition of its successor \( l_b \). The precondition at label \( l_b \) implies the weakest precondition of its two successors \( l_c \) and \( l_e \). The translation for \( e_2 \) establishes \( \text{shift}(Q) \land P[(\text{false} \lor s(0)]s(0)) \) which is the precondition of its successor \( l_d \). Therefore, the translation for \( e_1 \text{ or else } e_2 \) is valid.

### 7.2.3 Proof Translation of Instructions

In this section, we extend the translation function \( \nabla_S \) presented in Section 6.4 to rescue clauses, and once routines.

**Rescue clause**

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

\[
T_{S_1} \equiv \begin{cases} I_r \end{cases} s_1 \begin{cases} Q_n, Q_e \end{cases} \quad T_{S_2} \equiv \begin{cases} Q_e \end{cases} s_2 \begin{cases} (\text{Retry} \Rightarrow I_r) \land (\neg \text{Retry} \Rightarrow R_e), R_e \end{cases}
\]

First, the instruction \( s_1 \) is translated to a .try block. The exception label is updated to \( l_e \) because if an exception occurs in \( s_1 \), control will be transferred to the catch block at \( l_e \). Then, the instruction \( s_2 \) is translated into a catch block. For this, the exception object is first stored in a temporary variable and then \( s_2 \) is translated. In this translation, the Retry label is updated to \( l_{\text{start}} \) (the beginning of the routine). Finally, between labels \( l_e \) and \( l_i \), control is transferred to \( l_{\text{start}} \) if Retry is true; otherwise, the exception is pushed on top of the stack and re-thrown. The definition is:
\[
\bigg| T_{S_1} \quad T_{S_2} \bigg| \begin{array}{c}
\{ P \} \quad \text{do } s_1 \text{ rescue } s_2 \quad \{ Q_n, R_e \}
\end{array}
\] = \\
\text{.try{ }
\begin{align*}
\nabla_S \ (T_{S_1}, l_{\text{start}}, l_{\text{next}}, l_{\text{exc}}) &= \nabla_S \ (T_{S_2}, l_{\text{d}}, l_{\text{e}}, l_{\text{exc}}) \\
Q_n &\equiv \{ Q_n \} \\
Q_e &\equiv \{ Q_e \}
\end{align*}
\text{.catch } \text{System.Exception } \{ \\
\begin{align*}
l_{e} &\equiv \text{stloc } \text{last_exception} \\
l_{b} &\equiv \text{leave } l_{\text{next}} \\
l_{c} &\equiv \text{ldloc } \text{Retry} \\
l_{f} &\equiv \text{brfalse } l_{b} \\
l_{g} &\equiv \text{br } l_{\text{start}} \\
l_{h} &\equiv \text{idloc } \text{Retry} \\
l_{i} &\equiv \text{ldloc } \text{last_exception} \\
l_{l} &\equiv \text{rethrow}
\end{align*}
\}
\]

The bytecode for \text{s}_1 establishes \( Q_n \), which is the precondition of \( l_b \). The instruction at label \( l_b \) establishes \( Q_n \), which implies the precondition of its successor. The bytecode for \text{s}_2 establishes \( \text{Retry } \Rightarrow I_r \land \neg \text{Retry } \Rightarrow R_e \), which implies \( l_e \)'s precondition. Finally, it is easy to see that the instruction specifications of \( l_f, l_g, l_h, \) and \( l_i \) are valid (by applying the definition of weakest precondition).

**Once functions**

The translation of once functions uses the variables \( \text{T@m\_done}, \text{T@m\_exc}, \) and \( \text{T@m\_result} \). Let \( P \) be the following precondition, where \( T\_M\_RES \) is a logical variable:

\[
P \equiv \left\{ \begin{array}{c}
(\neg \text{T@m\_done} \land P') \\
(\text{T@m\_done} \land P'' \land \text{T@m\_result } = \text{T\_M\_RES} \land \neg \text{T@m\_exc} ) \lor \\
(\text{T@m\_done} \land P'' \land \text{T@m\_exc})
\end{array} \right\}
\]

and let \( Q_n' \) and \( Q_e' \) be the following postconditions:

\[
Q_n' \equiv \left\{ \begin{array}{c}
\text{T@m\_done } \land \neg \text{T@m\_exc } \land \\
(Q_n \lor ( P'' \land \text{Result } = \text{T\_M\_RES} \land \text{T@m\_result } = \text{T\_M\_RES} ))
\end{array} \right\}
\]

\[
Q_e' \equiv \left\{ \begin{array}{c}
\text{T@m\_done } \land \text{T@m\_exc } \land (Q_e \lor P'')
\end{array} \right\}
\]
Let $T_{\text{body}}$ be the following proof tree:

\[
T_{\text{body}} \equiv \frac{\text{Tree}_1}{\mathcal{A}, \{P\} \ T@m \ \{Q'_n, \ Q'_e\} \vdash}
\]

\[
\{ P'[\text{false}/T@m._\text{done}] \land T@m._\text{done} \} \quad \text{body}(T@m) \quad \{ Q_n, \ Q_e \}
\]

In this translation, the variable $T@m._\text{done}$ is evaluated at label $l_{\text{start}}$. If its value is $true$, control is transferred at the end of the function (label $l_k$). If $T@m._\text{done}$ evaluates to $false$, this variable is set to $true$, and the body of the function is translated using the translation function $\nabla_S$. This translation assumes that the variable $T@m._\text{result}$ is updated whenever the body of the function assigns to $\text{Result}$. If the body of the function triggers an exception, the exception is caught by the $\text{catch}$ block at labels $l_i - l_k$. The translation is defined as follows:

\[
\nabla_S \left( \frac{T_{\text{body}}}{\mathcal{A} \vdash \{ P \} \ T@m \ \{ Q'_n, \ Q'_e \}}, \ l_{\text{start}}, \ l_{\text{next}}, \ l_{\text{exc}} \right) =
\]
7.2 Proof Translator

\[
\begin{align*}
\{ P \} \\
\{ & \text{shift}(P) \land s(0) = T@m\_done \} \\
\{ & P' \land \neg T@m\_done \} \\
\{ & \text{shift}(P') \land \neg T@m\_done \land s(0) = true \} \\
\end{align*}
\]

\[
\begin{align*}
.l_a : & \text{ldsfld } T@m\_done \\
.l_b : & \text{brtrue } l_m \\
.l_c : & \text{ldc true} \\
.l_d : & \text{stsfld } T@m\_done \\
.l_f : & \text{idloc } result \\
.l_g : & \text{stsfld } T@m\_result \\
.l_h : & \text{leave } l_m \\
.l_i : & \text{ldloc } result \\
.l_j : & \text{stsfld } T@m\_exc \\
.l_k : & \text{stloc } last\_exc \\
.l_l : & \text{stsfld } T@m\_exc \\
.l_m : & \text{ldsfld } T@m\_exc \\
.l_n : & \text{brfalse } l_q \\
.l_o : & \text{ldsfld } last\_exc \\
.l_p : & \text{throw} \\
.l_q : & \text{ldsfld } T@m\_result \\
.l_r : & \text{ret}
\end{align*}
\]
where \( Q' \equiv \{ \begin{align*}
& T@m\_done \land \\
& (Q_n \lor \\
& P'' \land T@m\_result = T@m\_RESULT \land \\
& T@m\_exc = \text{false} \\
& P''' \land T@m\_exc = \text{true} 
\} \} \) 

### 7.3 Applications

In this section, we show an example of the translation of Eiffel proofs. The source proof is presented in Section 3.2.4. This example implements an integer division, which always terminates normally. If the second operand is zero, it returns the first operand; otherwise the result is equal to the integer division \( x/y \). The Eiffel proof-transforming compiler takes that proof, and generates a derivation in the bytecode logic. Similar to the example presented in Section 6.6, the PTC starts with the translation of the class rule. Then, the compiler generates the translation of the routine implementation rule, and finally it applies the instruction translation. The bytecode derivation is the following:

\[
\{ \tau(\text{Current}) \prec \text{MATH} \land \text{true} \} \quad \text{MATH:safe\_division} \quad \{ Q, \text{false} \} \\
\{ \tau(\text{Current}) = \text{MATH} \land \text{true} \} \quad \text{impl(MATH, safe\_division)} \quad \{ Q, \text{false} \} \\
\{ \tau(\text{Current}) \preceq \text{MATH} \land \text{true} \} \quad \text{MATH:safe\_division} \quad \{ Q, \text{false} \}
\]

where 
\[ Q \equiv (y = 0 \Rightarrow \text{Result} = x) \land (y/ = 0 \rightarrow \text{Result} = x/y) \]

The triple \( \{ \tau(\text{Current}) \prec \text{MATH} \land \text{true} \} \quad \text{MATH:safe\_division} \quad \{ Q, \text{false} \} \) is proved using the false axiom due to the class MATH has no descendants. Since the routine safe\_division is implemented in the class MATH, then impl(MATH, safe\_division) = MATH@safe\_division. The second hypothesis is obtained by the translation of the routine implementation rule. The proof in CIL is as follows:

\[
\{ \text{true} \} \quad \text{body(MATH@safe\_division)} \quad \{ Q, \text{false} \} \\
\{ \text{true} \} \quad \text{MATH@safe\_division} \quad \{ Q, \text{false} \}
\]

Figure 7.1 illustrates the translation of the body of the routine safe\_division. This example is the result produced by the translation \( \nabla S \) of the example proof presented in Figure 3.9. The generated bytecode for the body of the routine is enclosed in a try block (lines IL001 to IL007). Since the routine always terminates normally, the precondition of the instructions at labels IL017 and IL018 is false.
\begin{align*}
\{(y \neq 0 \land z = 0) \lor (y = 0 \land (z = 1 \lor z = 0))\} \\
\{ (y \neq 0 \land z = 0) \lor (y = 0 \land (z = 1 \lor z = 0)) \land s(0) = x \} \\
\{ (y \neq 0 \land z = 0) \lor (y = 0 \land (z = 1 \lor z = 0)) \land s(1) = x \land s(0) = y \} \\
\{ (y \neq 0 \land z = 0) \lor (y = 0 \land (z = 1 \lor z = 0)) \land s(1) = x \land s(1) = y \land s(0) = z \} \\
\{ (y \neq 0 \land z = 0) \lor (y = 0 \land (z = 1 \lor z = 0)) \land s(1) = x \land s(0) = y + z \} \\
\{ (y \neq 0 \land z = 0) \lor (y = 0 \land (z = 1 \lor z = 0)) \land s(0) = x / (y + z) \} \\
\{ (y = 0 \Rightarrow Result = x) \land (y \neq 0 \Rightarrow Result = x/y) \} \\
\{ y = 0 \land z = 0 \land excV \neq null \land s(0) = excV \} \\
\{ y = 0 \land z = 0 \} \\
\{ y = 0 \land z = 0 \land s(0) = 1 \} \\
\{ y = 0 \land z = 1 \} \\
\{ y = 0 \land z = 1 \land s(0) = true \} \\
\{ \neg Retry \Rightarrow false \land Retry \Rightarrow (y = 0 \lor z = 1) \} \\
\{ \neg Retry \Rightarrow false \land \neg Retry \Rightarrow (y = 0 \lor z = 1) \land s(0) = Retry \} \\
\{ y = 0 \lor z = 1 \} \\
\{ false \} \\
\{ false \land s(0) = last_exception \} \\
\{ (y = 0 \Rightarrow Result = x) \land (y \neq 0 \Rightarrow Result = x/y) \} \\
\{ (y = 0 \Rightarrow Result = x) \land (y \neq 0 \Rightarrow Result = x/y) \land s(0) = Result \}
\end{align*}

\texttt{try \{} \\
\texttt{IL001 : ldloc x} \\
\texttt{IL002 : ldloc y} \\
\texttt{IL003 : ldloc z} \\
\texttt{IL004 : binop\texttt{+}} \\
\texttt{IL005 : binop//} \\
\texttt{IL006 : stloc Result} \\
\texttt{IL007 : leave IL019} \\
\texttt{catch System.Exception \{} \\
\texttt{IL009 : stloc last\texttt{-exception}} \\
\texttt{IL010 : ldc 1} \\
\texttt{IL011 : stloc z} \\
\texttt{IL012 : ldc true} \\
\texttt{IL013 : stloc Retry} \\
\texttt{IL014 : ldloc Retry} \\
\texttt{IL015 : brfalse IL017} \\
\texttt{IL016 : br 01} \\
\texttt{IL017 : ldloc last\texttt{-exception}} \\
\texttt{IL018 : rethrow} \\
\texttt{\}} \\
\texttt{IL019 : ldloc Result} \\
\texttt{IL020 : ret}

Fig. 7.1: Bytecode Proof generated by the PTC from the Example Proof of Figure 3.9.
7.4 Soundness of the Contact Translator

The soundness theorem of the instruction translator has been presented on Section 6.7. In this section, we describe the soundness theorem of the contract translator. The theorem expresses: given two heaps and a state, if the expression \( e \) is well-formed then the value of the translation of the expression \( e \) is equal to the value returned by the evaluation of \( e \). The functions \( \text{value}_C \), \( \text{value}_T \), \( \text{value}_{\text{Exp}} \), \( \text{value}_{\text{Call}} \) and \( \text{value}_{\text{Arg}} \) evaluate boolean expressions, Eiffel types, expressions, routine calls and arguments respectively. The theorem is the following:

**Theorem 5** (Soundness of the Eiffel Contract Translator).

\[
\forall b : \text{BoolExp}, t : \text{TypeFunc}, e : \text{Expr}, c : \text{CallRoutine}, p : \text{Argument} : \\
(\text{wellF}_C b) \Rightarrow (\text{value}_C b h_1 h_2 s) = (\nabla_C b) h_1 h_2 s \\
(\text{wellF}_T t) \Rightarrow (\text{value}_T t h_1 h_2 s) = (\nabla_T t) h_1 h_2 s \\
(\text{wellF}_{\text{Exp}} e) \Rightarrow (\text{value}_{\text{Exp}} e h_1 h_2 s) = (\nabla_{\text{Exp}} e) h_1 h_2 s \\
(\text{wellF}_{\text{Call}} c) \Rightarrow (\text{value}_{\text{Call}} c h_1 h_2 s) = (\nabla_{\text{Call}} c) h_1 h_2 s \\
(\text{wellF}_{\text{Arg}} p) \Rightarrow (\text{value}_{\text{Arg}} p h_1 h_2 s) = (\nabla_{\text{Arg}} p) h_1 h_2 s)
\]

The soundness proof of the specification translator has been formalized and proved in Isabelle. The proof runs by induction on the syntactic structure of the expression. The full proofs can be found in our technical report [95].
Chapter 8

Java-Specific Aspects of Proof-Transforming Compilation

Proof-transforming compiles are a powerful approach to translate proofs. Recent works [9, 96, 80, 103] have shown that proofs can be translated from object-oriented languages to bytecode. If the source and the target language are close, the proof translation is simple, for example the proof translation presented in Chapter 6. However, if these languages are not close, the translation can be hard. One example is the translation from Eiffel to CIL, which is described in Chapter 7. The main difficulties in the Eiffel proof-transforming compiler are the translation of contracts and the translation of exception handling.

Although our proof-transforming compiler from Java to bytecode has a straightforward contract translation, this translation is also hard. The main difficult comes from the translation of try-catch, try-finally, and break instructions. If one uses a bytecode language similar to CIL, this translation is simpler than using Java Bytecode. The main problem translating proofs to Java Bytecode is the formalization of the compilation function, and the soundness proof of the translation.

In Java Bytecode, a try-finally instruction is compiled using code duplication: the finally block is put after the try block. If try-finally instructions are used inside of a while loop, the compilation of break instructions first duplicates the finally blocks and then inserts a jump to the end of the loop. Furthermore, the generation of exception tables is also harder. The code duplicated before the break may have exception handlers different from those of the enclosing try block. Therefore, the exception table must be changed so that exceptions are caught by the appropriate handlers.

In this chapter, we present a proof-transforming compiler for a subset of Java, which handles try-catch, try-finally, and break instructions. In Section 8.1 we present the proof translation using CIL bytecode. In Section 8.2 we show the proof translation using Java Bytecode.

This chapter is partially based on the published work [80].
8.1 Proof Translation using CIL

This section presents proof-transformation for a subset of Java using CIL. We assume break instructions are not used inside of a try-finally instruction. This assumption simplifies the translation function and the soundness proof. In Section 8.2, we remove this assumption and we formalize the translation of try-catch, try-finally, and break instructions using Java Bytecode.

8.1.1 Translation Basics

The Java proof-transforming compiler is based on two translation functions, \( \nabla_S \) and \( \nabla_E \) for instructions and expressions respectively. The translation of expression is done with the same function \( \nabla_E \) presented in Section 6.3. To be able to handle the Java exception handling, the translation function \( \nabla_S \) also takes a mapping function that maps exception types to labels. Given an exception type \( T \), this mapping yields the label where the exception of type \( T \) is caught. The signature of these functions are defined as follows:

\[
\nabla_E : \text{Precondition} \times \text{Expression} \times \text{Postcondition} \times \text{Label} \rightarrow \text{BytecodeProof} \\
\nabla_S : \text{ProofTree} \times \text{Label} \times \text{Label} \times \text{Label} \times [\text{ExcType} \rightarrow \text{Label}] \rightarrow \text{BytecodeProof}
\]

ProofTree is a derivation in the logic presented in Section 3.3. The labels are: (1) \( l_{\text{start}} \) for the first label of the resulting bytecode; (2) \( l_{\text{next}} \) for the label after the resulting bytecode; (3) \( l_{\text{break}} \) for the jump target for break instructions. ExcType is an exception type (a descendent type of the type Exception). As a convention, given the mapping function \( m : \text{ExcType} \rightarrow \text{Label} \), \( m[T] \) yields the label where the exception \( T \) is caught. The application \( m[T \rightarrow l] \) updates the mapping function \( m \) where the value of the parameter \( T \) is replaced by \( l \).

In the following sections, we present the translation for compound, while, break, throw, try-catch, and try-finally instructions. The translation of basic instructions such as assignment, if then else, and loops is similar to the translation presented in Section 6.4.

8.1.2 Compound

The translation of the compound for Java is similar to the translation presented in Section 6.4.2. Let \( T_{S_1} \) and \( T_{S_2} \) be the following proof trees:

\[
T_{S_1} \equiv \begin{cases} \text{Tree}_1 \{ P \} \ s_1 \ { Q_n, R_b, R_e } \\ \end{cases} \quad T_{S_2} \equiv \begin{cases} \text{Tree}_2 \{ Q_n \} \ s_2 \ { R_n, R_b, R_e } \\ \end{cases}
\]
The definition of the translation is the following:

\[
\nabla_S \left( \frac{T_{S_1}}{P} \frac{T_{S_2}}{s_1; s_2} \begin{array}{c} T_{\text{pr}} \\ \{ R_n, R_b, R_e \} \end{array}, \ l_{\text{start}}, l_{\text{next}}, l_{\text{break}}, m \right) = \\
\nabla_S (T_{S_1}, l_{\text{start}}, l_b, l_{\text{break}}, m) \\
\nabla_S (T_{S_2}, l_b, l_{\text{next}}, l_{\text{break}}, m)
\]

In this translation, the mapping function \( m \) is used as an argument in the translation of \( s_1 \) and \( s_2 \). The bytecode is valid because the bytecode for \( s_1 \) establishes \( Q_n \), which is the precondition of the first instruction of the bytecode for \( s_2 \).

### 8.1.3 While Rule

Let \( T_{S_1} \) and \( T_{\text{while}} \) be the following proof trees:

\[
T_{S_1} \equiv \frac{T_{\text{pr}}}{e \land I} s_1 \begin{array}{c} I, Q_b, R_e \end{array}
\]

\[
T_{\text{while}} \equiv \frac{T_{S_1}}{I} \begin{array}{c} \text{while} \ (e) s_1 \begin{array}{c} (I \land \neg e) \lor Q_b, false, R_e \end{array} \end{array}
\]

In this translation, first the loop expression is evaluated at \( l_c \). If this expression evaluates to \( true \), control is transferred to \( l_b \), the start label of the loop body. In the translation of \( T_{S_1} \), the start label and next labels are updated with the labels \( l_b \) and \( l_c \). Furthermore, the break label is updated with the label at the end of the loop (\( l_{\text{next}} \)). The definition of the translation is the following:

\[
\nabla_S (T_{\text{while}}, l_{\text{start}}, l_{\text{next}}, l_{\text{break}}, m) = \\
\{I\} \quad l_a : \text{br} \quad l_c \\
\nabla_S (T_{S_1}, l_b, l_c, l_{\text{next}}, m) \\
\nabla_E (I, e, (\text{shift}(I) \land s(0) = e), l_{\text{c}}) \\
\{\text{shift}(I) \land s(0) = e\} \quad l_d : \text{brtrue} \quad l_b
\]
The instruction \( \text{br} \) establishes \( I \), which is the precondition of the successor instruction (the first instruction of the translation of \( s_1 \)). The translation of \( s_1 \) establishes \( I \), which implies the precondition of its successor. The translation of \( e \) establishes

\[
\text{shift}(I) \land s(0) = e
\]
because the evaluation of the expression pushes the result on top of the stack. This postcondition implies the precondition of the successor instruction \( b_{\text{true}} \). Finally, the instruction \( b_{\text{true}} \) establishes the preconditions of both possible successor instructions, namely \( e \land I \) for the successor \( l_b \) (the first instruction of \( s_1 \)), and \( I \land \neg e \) for \( l_{\text{next}} \). Therefore, the produced bytecode proof is valid.

### 8.1.4 Break Rule

The translation of the \( \text{break} \) instruction transfers control to the end of the loop. The label that represents the end of the loop is stored in the parameter \( l_{\text{break}} \). Thus, the \( \text{break} \) instruction is translated using a \( \text{br} \) instruction. The definition of the translation is the following:

\[
\nabla_s \left( \{ P \} \text{break} \{ \text{false}, P, \text{false} \}, l_{\text{start}}, l_{\text{next}}, l_{\text{break}}, m \right)
\]

\[
\{ P \} \l_\text{start} : \text{br} l_{\text{break}}
\]

To argue that the bytecode proof is valid, we have to show that the postcondition \( P \) implies the precondition of \( l_{\text{break}} \). This can be proven using the hypothesis of the soundness theorem presented in Section 8.1.8.

### 8.1.5 Throw Rule

The \( \text{throw} \) instruction is translated using the CIL \( \text{athrow} \) instruction. The translation first pushes the expression \( e \) on top of the stack, and then it adds the instruction \( \text{athrow} \). We assume that the instruction \( \text{athrow} \) takes the expression \( e \) on the top of the stack, then assigns it to the variable \( \text{excV} \), and finally control is transferred to the label where the exception is caught. Here, the mapping function \( m \) is used to prove soundness (the proof is presented in Appendix D). The translation is defined as follows:
8.1 Poof Translation using CIL

$$\forall\left(\begin{array}{c}
\{P[e/\text{excV}]\} \\
\text{throw } e \{\text{false, false, } P\}
\end{array}\right), \text{lstart, lnext, lbreak, m}\right) =$$

$$\forall\left(P[e/\text{excV}], e, (\text{shift}(P[e/\text{excV}]) \land s(0) = e), \text{lstart} \right)$$

$$\{\text{shift}(P[e/\text{excV}]) \land s(0) = e\} \text{l}_b : \text{athrow}$$

The translation for $e$ establishes $\text{shift}(P[e/\text{excV}]) \land s(0) = e$ because it pushes the expression $e$ on top of the stack. This postcondition implies the precondition at label $l_b$. Therefore, the translation is valid.

8.1.6 Try-catch Rule

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

$$T_{S_1} \equiv \begin{array}{c}
\text{Tree}_1 \\
\{P\} \quad s_1 \{Q_n, Q_b, Q\}
\end{array}$$

$$T_{S_2} \equiv \begin{array}{c}
\text{Tree}_2 \\
\{Q'_e[e/\text{excV}]\} \quad s_2 \{Q_n, Q_b, R_e\}
\end{array}$$

where

$$Q \equiv ( (Q'_e \land \tau(\text{excV}) \not\preceq T) \lor (Q'_e \land \tau(\text{excV}) \preceq T) )$$

Let $R$ be the following postcondition:

$$R \equiv (R_e \lor (Q''_e \land \tau(\text{excV}) \not\preceq T))$$

To translate try-catch instructions, we use the CIL instructions .try and catch. First, the instruction $s_1$ is translated in the body of .try. This translation updates the mapping function $m$, where the label of the exception type $T$ is changed by $l_c$. This update is done to express that the exceptions of type $T$ are caught at the label $l_c$. If $s_1$ triggers an exception, control is transferred to the catch block. In this catch block, first the exception value is stored in $e$ (at label $l_c$), and then the translation of $s_2$ is added. The definition of the translation is:
The bytecode for the instruction $s_1$ establishes $Q_n$, which is the precondition of the successor instruction $l_b$. The precondition at $l_b$, $Q_n$, implies the precondition of its successor $l_{next}$. The precondition at label $l_c$ is established by the translation of $s_1$. Finally, the translation of $s_2$ establishes $Q_n$, which is the precondition at label $l_e$. Therefore, the translation is valid.

### 8.1.7 Try-finally Rule

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

$$T_{S_1} \equiv \frac{T_{S_1}}{Tree_1}$$

$$T_{S_2} \equiv \frac{T_{S_2}}{Tree_2}$$
where
\[ Q \equiv \left( (Q_n \land X\text{Tmp} = \text{normal}) \lor (Q_b \land X\text{Tmp} = \text{break}) \lor \left( Q_e[e\text{Tmp}/\text{excV}] \land X\text{Tmp} = \text{exc} \land e\text{Tmp} = \text{excV} \right) \right) \]

and
\[ R \equiv \left( (R'_n \land X\text{Tmp} = \text{normal}) \lor (R'_b \land X\text{Tmp} = \text{break}) \lor \left( R'_e[e\text{Tmp}/\text{excV}] \land X\text{Tmp} = \text{exc} \right) \right) \]

The try-finally instruction is translated using .try and .finally instructions. In the body of the .try instruction, first, the instruction \( s_1 \) is translated using \( \nabla_S \), and then the instruction leave is added. In the finally block, the instruction \( s_2 \) is translated using \( \nabla_S \).

The definition of the translation is:

\[
\nabla_S \left( \begin{array}{c}
T_{S_1} \\
T_{S_2} \\
\{ P \} \\
\text{try } s_1 \text{ finally } s_2 \\
\{ R'_n, R'_b, R'_e \}
\end{array} \right), \ l_{\text{start}}, l_{\text{next}}, l_{\text{break}}, m =
\]

..try {
    \nabla_S (T_{S_1}, l_{\text{start}}, l_b, l_{\text{break}}, m)
    \{ Q_n \} \ l_b : \text{leave} \ l_{\text{next}}
}
..finally {
    \nabla_S (T_{S_2}, l_c, l_d, l_{\text{break}}, m)
    \{ R \} \ l_d : \text{endfinally}
}

### 8.1.8 Soundness Theorem

In this section, we extend the soundness theorem of the instruction translator presented in Section 6.7. The translation described in the above sections handles break instructions and the Java exception handling mechanism. This soundness theorem adds hypothesis that relates the break postcondition with the preconditions where the break is translated, and the exceptional postcondition with the precondition where the exception is caught.

The theorem states that if (1) we have a valid source proof for the instruction \( s_1 \), and (2) we have a proof translation from the source proof that produces the instructions
Theorem 6 (Soundness of the Java Instruction Translator).

\[ \vdash Tree_1 \underbrace{\{ P \} \ s_1 \ \{ Q_n, Q_b, Q_e \}}_{s} \land \]
\[ (I_{start} \ldots I_{end}) = \nabla_S \left( \underbrace{Tree_1 \underbrace{\{ P \} \ s_1 \ \{ Q_n, Q_b, Q_e \}}_{s} \land \ l_{start}, l_{end+1}, l_{break}, m \right) \land \]
\[ (Q_n \Rightarrow E_{l_{end+1}}) \land \]
\[ (Q_b \Rightarrow E_{l_{break}}) \land \]
\[ (\forall T : Type : T \leq Throwable : (Q_e \land excV \neq null \land s(0) = excV) \Rightarrow E_{m[T]}) \]
\[ \Rightarrow \forall l \in l_{start} \ldots l_{end} \vdash \{ E_l \} I_l \]

The full proofs can be found in Appendix D. The proof runs by induction on the structure of the derivation tree for \( \{ P \} s_1 \{ Q_n, Q_e \} \).

8.2 Proof Translation using Java Bytecode

In Chapter 6 and Chapter 7 we have described proof-transforming compilation for a core object-oriented language and for a subset of Eiffel respectively. These translations generate proofs in a logic for CIL. These PTCs can be easily adapted to generate other bytecode such as JVM. The main change is the use of exception tables instead of try catch instructions. However, this is not the case for the translation presented in Section 8.1.

The main difference between JVM and CIL is the exception handling mechanism. In CIL, exceptions are handled using a similar mechanism to C# and Java: try and catch instructions. To handle exceptions, JVM uses exception tables. To compile try-finally instructions, the Java compiler uses code duplication. Consider the following example:
8.2 Proof Translation using Java Bytecode

while (i < 20) {
    try {
        try {
            try {
                ... break; ...
            }
            catch (Exception e) {
                i = 9;
            }
        }
        finally {
            throw new Exception();
        }
        catch (Exception e) {
            i = 99;
        }
    }
}

The finally body is duplicated before the break. But the exception thrown in the finally block must be caught by the outer try-catch. To achieve that, the compiler creates, in the following order, exception lines for the outer try-catch, for the try-finally, and for the inner try-catch. When the compiler reaches the break, it divides the exception entry of the inner try-catch and try-finally into two parts so that the exception is caught by the outer try-catch.

This code duplication increases the complexity of the compilation and translation functions, especially the formalization and its soundness proof. In the following, we present the proof translation using a similar language to Java Bytecode. In particular, we use the CIL instructions names instead of the JVM (for example we write ldloc instead of pushv).

8.2.1 Translation Basics

The instruction translation is done using the function \( \nabla_S \). The function \( \nabla_S \) generates a bytecode proof and an exception table from a source proof. This function is defined as a composition of the translations of its sub-trees. The signature is the following:

\[
\nabla_S : \text{ProofTree} \times \text{Label} \times \text{Label} \times \text{Label} \times \text{List}[\text{Finally}] \times \text{ExcTable} \rightarrow [\text{BytecodeProof} \times \text{ExcTable}]
\]

ProofTree is a derivation in the source logic. The three labels are: (1) \( l_{\text{start}} \) for the first label of the resulting bytecode; (2) \( l_{\text{next}} \) for the label after the resulting bytecode; (3) \( l_{\text{break}} \) for the jump target for break instructions.
The BytecodeProof type is defined as a list of InstrSpec, where InstrSpec is an instruction specification. The Finally type, used to translate finally instructions, is defined as a tuple [ProofTree, ExcTable]. Furthermore, the function $\nabla_S$ takes an exception table as parameter and produces an exception table. This is necessary because the translation of break instructions can lead to a modification of the exception table as described above.

The ExcTable type is defined as follows:

$$\text{ExcTable} := \text{List}[\text{ExcTableEntry}]$$

$$\text{ExcTableEntry} := [\text{Label}, \text{Label}, \text{Label}, \text{Type}]$$

In the ExcTableEntry type, the first label is the starting label of the exception line, the second denotes the ending label, and the third is the target label. An exception of type $T_1$ thrown at line $l$ is caught by the exception entry $[l_{\text{start}}, l_{\text{end}}, l_{\text{arg}}, T_2]$ if and only if $l_{\text{start}} \leq l < l_{\text{end}}$ and $T_1 \preceq T_2$. Control is then transferred to $l_{\text{arg}}$.

In the following, we present the proof translation using Java Bytecode for compound, while, try-finally, and break.

### 8.2.2 Compound

Let $T_S_1$ and $T_S_2$ be the following proof trees:

$$T_S_1 = \frac{T_{\text{tree}_1}}{\{ P \} \ \{ s_1 \}} \ \{ Q_n, R_b, R_e \}$$

$$T_S_2 = \frac{T_{\text{tree}_2}}{\{ Q_n \} \ \{ s_2 \}} \ \{ R_n, R_b, R_e \}$$

In the translation of $T_S_1$, the label $l_{\text{next}}$ is the start label of the translation of $s_2$, say $l_b$. The translation of $T_S_2$ uses the exception table produced by the translation of $T_S_1$, $et_1$. The translation of $T_{S_1;S_2}$ yields the concatenation of the bytecode proofs for the sub-instructions and the exception table produced by the translation of $T_S_2$.

Let $[B_{S_1}, et_1]$ and $[B_{S_2}, et_2]$ be of type [BytecodeProof, ExcTable], and defined as follows:

$$[B_{S_1}, et_1] = \nabla_S (T_S_1, l_{\text{start}}, l_b, l_{\text{break}}, f, et)$$

$$[B_{S_2}, et_2] = \nabla_S (T_S_2, l_b, l_{\text{next}}, l_{\text{break}}, f, et_1)$$

The translation is defined as follows:

$$\nabla_S \left( \frac{T_S_1}{\{ P \} \ \{ s_1; s_2 \}} \frac{T_S_2}{\{ R_n, R_b, R_e \} \ \{ l_{\text{start}}, l_{\text{next}}, l_{\text{break}}, f, et \} \ \{ B_{S_1}, B_{S_2}, et_1 \} \ \{ et_2 \} } \right) =$$

The bytecode for $s_1$ establishes $Q_n$, which is the precondition of the first instruction of the bytecode for $s_2$. Therefore, the concatenation $B_{S_1} + B_{S_2}$ produces a sequence of valid instruction specifications. We will formalize soundness in Section 8.2.6.
8.2 Proof Translation using Java Bytecode

8.2.3 While Rule

Let $T_S$ be the following proof tree:

$T_S \equiv \frac{\text{Tree}}{\{ e \land I \}} \quad s_1 \quad \{ I, Q_b, R_e \}$

In this translation, the finally list is set to the empty set, because a break instruction inside the loop jumps to the end of the loop without executing any finally block.

Let $b_{br}$ and $b_{brtrue}$ be instruction specifications, and let $B_{S_1}$ and $B_e$ be bytecode proofs defined as follows:

$$b_{br} = \{ I \} \quad l_b : \text{br } l_c$$

$$[B_{S_1}, et_1] = \nabla_S (T_{S_1}, l_b, l_c, l_{next}, \emptyset, et)$$

$$B_e = \nabla_E (I, e, (\text{shift}(I) \land s(0) = e), l_c)$$

$$b_{brtrue} = \{ \text{shift}(I) \land s(0) = e \} \quad l_d : \text{brtrue } l_b$$

The definition of the translation is the following:

$$\nabla_S \left( \frac{T_S}{\{ I \}} \quad \text{while } (e) \quad s_1 \quad \{ (I \land \neg e) \lor Q_b, false, R_e \} \quad l_{start}, l_{next}, l_{break}, f, et \right) =$$

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

The instruction $b_{br}$ establishes $I$, which is the precondition of the successor instruction (the first instruction of $B_{S_1}$). $B_{S_1}$ establishes $I$, which implies the precondition of its successor $B_e$. $I$. $B_e$ establishes $\text{shift}(I) \land s(0) = e$ because the evaluation of the expression pushes the result on top of the stack. This postcondition implies the precondition of the successor instruction $b_{brtrue}$. The instruction specification $b_{brtrue}$ establishes the preconditions of both possible successor instructions, namely $e \land I$ for the successor $l_b$ (the first instruction of $B_{S_1}$), and $I \land \neg e$ for $l_{next}$. Therefore, the produced bytecode proof is valid.

8.2.4 Try-Finally Rule

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

$$T_{S_1} \equiv \frac{\text{Tree}}{\{ P \}} \quad s_1 \quad \{ Q_n, Q_b, Q_e \}$$

$$T_{S_2} \equiv \frac{\text{Tree}}{\{ Q \}} \quad s_2 \quad \{ R, R_b', R_e' \}$$
where
\[ Q \equiv \left( (Q_n \land X Tmp = \text{normal}) \lor (Q_b \land X Tmp = \text{break}) \lor \left( \left( Q_e[Ump/\text{excV}] \land X Tmp = \text{exc} \land Ump = \text{excV} \right) \right) \right) \]
\[ R \equiv \left( (R'_n \land X Tmp = \text{normal}) \lor (R'_b \land X Tmp = \text{break}) \lor \left( \left( R'_e[Ump/\text{excV}] \land X Tmp = \text{exc} \right) \right) \right) \]

In this translation, the bytecode for \( s_1 \) is followed by the bytecode for \( s_2 \). In the translation of \( T_{S_1} \), the `finally` block is added to the finally-list \( f \) with \( T_{S_2} \)'s source proof tree and its associated exception table. The corresponding exception table is retrieved using the function \( \text{getExcLines} \). The signature of this function is as follows:

\[ \text{getExcLines} : \text{Label} \times \text{Label} \times \text{ExcTable} \rightarrow \text{ExcTable} \]

Given two labels and an exception table \( et \), \( \text{getExcLines} \) returns, per every exception type in \( et \), the first \( et \)'s exception entry (if any) for which the interval made by the starting and ending labels includes the two given labels.

Then, the translation of \( s_1 \) adds a new exception entry for the `finally` block, to the exception table \( et \). Finally, the bytecode proof for the case when \( s_1 \) throws an exception is created. The exception table of this translation is produced by the predecessor translations.

Let \( et_1 \) and \( et' \) be the following exception tables:

\[ et_1 = et + [l_{\text{start}}, l_b, l_d, \text{any}] \]
\[ et' = \text{getExcLines}(l_{\text{start}}, l_b, et_1) \]

Let \( b_{br} \), \( b_{stloc} \), \( b_{ldloc} \), and \( b_{athrow} \) be bytecode proofs, and let \( B_{S_1} \), \( B_{S_2} \), and \( B'_{S_2} \) be bytecode proofs:

\[ [B_{S_1}, et_2] = \nabla_S (T_{S_1}, l_{\text{start}}, l_b, l_{\text{break}}, [T_{S_2}, et'] + f, et_1) \]
\[ [B_{S_2}, et_3] = \nabla_S (T_{S_2}, l_b, l_c, l_{\text{break}}, f, et_2) \]
\[ b_{br} = \begin{cases} \text{shift}(Q_e) \land \\ \text{excV} \neq \text{null} \land s(0) = \text{excV} \end{cases} \]
\[ b_{stloc} = \begin{cases} l_c : \text{br} \ l_{\text{next}} \end{cases} \]
\[ b_{ldloc} = \begin{cases} R'_n \lor R'_b \lor R'_e \end{cases} \]
\[ b_{athrow} = \begin{cases} (R'_n \lor R'_b \lor R'_e) \land s(0) = e Tmp \end{cases} \]
\[ b_{ldloc} = \begin{cases} l_d : \text{stloc} \ e Tmp \end{cases} \]
\[ b_{athrow} = \begin{cases} l_g : \text{athrow} \end{cases} \]
8.2 Proof Translation using Java Bytecode

The translation is defined as follows:

$$\nabla_S \left( \begin{array}{c}
T_{S_1} \\
T_{S_2}
\end{array} \right), \{ P \} \text{ try } s_1 \text{ finally } s_2 \{ R'_n, R'_b, R'_e \} = [ B_{S_1} + B_{S_2} + b_{\text{goto}} + b_{\text{pop}} + B_{S'_2} + b_{\text{pushv}} + b_{\text{athrow}}, et ]$$

It is easy to see that the instruction specifications $b_{\text{br}}, b_{\text{stloc}}, b_{\text{idloc}},$ and $b_{\text{athrow}}$ are valid (by applying the definition of the weakest precondition). However, the argument for the translation of $T_{S_1}$ and $T_{S_2}$ is more complex. Basically, the result is a valid proof because the proof tree inserted in $f$ for the translation of $T_{S_1}$ is a valid proof and the postcondition of each finally block implies the precondition of the next one. Furthermore, for normal execution, the postcondition of $B_{S_1}$ $(Q_n)$ implies the precondition of $B_{S_2}$ $(Q)$.

8.2.5 Break Rule

To specify the rules for $\text{break}$, we use the following recursive function:

$$\text{divide} : \text{ExcTable} \times \text{ExcTableEntry} \times \text{Label} \times \text{Label} \rightarrow \text{ExcTable}$$

The definition of $\text{divide}$ assumes that the exception entry is in the given exception table and the two given labels are in the interval made by the exception entry’s starting and ending labels. Given an exception entry $y$ and two labels $l_s$ and $l_e$, $\text{divide}$ compares every exception entry, say $x$, of the given exception table to $y$. If the interval defined by $x$’s starting and ending labels is included in the interval defined by $y$’s starting and ending labels, then $x$ must be divided to have the appropriate behavior of the exceptions. Thus, the first and the last interval of the three intervals defined by $x$’s starting and ending labels, $l_s$, and $l_e$ are returned, and the procedure is continued for the next exception entry. If $x$ and $y$ are equal, then recursion stops as $\text{divide}$ reached the expected entry. The formal definition of $\text{divide}$ is the following:
divide : ExcTable × ExcTableEntry × Label × Label → ExcTable
\[divide : (\text{[]} , e' , l_s , l_e) = [ e']\]
\[divide : (e : et , e' , l_s , l_e) = [ l_{\text{start}} , l_s , l_{\text{targ}} , T_1 ] + [ l_e , l_{\text{end}} , l_{\text{targ}} , T_1 ] +\]
\[divide(e , e' , l_s , l_e)\]
\[\text{if } e \subseteq e' \text{ and } e \neq e'\]
\[\text{if } e = e'\]
\[e : et\]
\[\text{otherwise}\]

where
\[e \equiv [l_{\text{start}} , l_{\text{end}} , l_{\text{targ}} , T_1]\] and \[e' \equiv [l'_{\text{start}} , l'_{\text{end}} , l'_{\text{targ}} , T_2]\]

\[\subseteq : \text{ExcTableEntry} \times \text{ExcTableEntry} \rightarrow \text{Boolean}\]
\[\subseteq : ([l_{\text{start}} , l_{\text{end}} , l_{\text{targ}} , T_1] , [l'_{\text{start}} , l'_{\text{end}} , l'_{\text{targ}} , T_2]) = \text{true}\]
\[\text{if } (l'_s \leq l_s) \land (l'_e \geq l_e)\]
\[\text{otherwise}\]

When a break instruction is encountered, the proof tree of every finally block the break has to execute upon exiting the loop is translated. Then, control is transferred to the end of the loop using the label \(l_{\text{break}}\). Let \(f_i = [T_{F_i} , et'_i]\) denote the \(i\)-th element of the list \(f\), where
\[T_{F_i} = \left\{ \begin{array}{l}
\text{Tree}_i \\
\{ U^i \} \\
\{ V^i \}
\end{array} \right\}
\]
and \(U^i\) and \(V^i\) have the following form, which corresponds to the Hoare rule for try–finally:
\[U^i \equiv \left\{ \begin{array}{l}
(U^i_n \land X Tmp = \text{normal}) \lor (U^i_b \land X Tmp = \text{break}) \lor \\
(U^i[eTmp/excV] \land X Tmp = \text{exc} \land eTmp = \text{excV})
\end{array} \right\}
\]
\[V^i \equiv \left\{ \begin{array}{l}
(V^i_n \land X Tmp = \text{normal}) \lor \\
(V^i_b \land X Tmp = \text{break}) \lor \\
(V^i = \text{exc})
\end{array} \right\}
\]

Let \(B_{F_i}\) be a BytecodeProof for \(T_{F_i}\) such that:
\[[B_{F_i} , et_{i+1}] = \nabla_S (T_{F_i} , l_{\text{start}+i} , l_{\text{start}+i+1} , l_{br} , f_{i+1} \cdots f_k , divide(et_i , et'_i , l_{\text{start}+i} , l_{\text{start}+i+1}))\]
\[b_{br} = \{B^i_k\} \ l_{\text{start}+k+1} : \text{br} \ l_{br}\]

The definition of the translation is the following:
\[\nabla_S \left( \begin{array}{l}
\{ P \} \break \{ \text{false} , P , \text{false} \} , l_{\text{start}} , l_{\text{next}} , l_{br} , f , et_0
\end{array} \right)
\]
\[= [ B_{F_1} + B_{F_2} + \ldots + B_{F_k} + b_{br} , et_k] \]
To argue that the bytecode proof is valid, we have to show that the postcondition of $B_F$ implies the precondition of $B_{F_{i+1}}$ and that the translation of every block is valid. This is the case because the source rule requires the break-postcondition of $s_1$ to imply the normal precondition of $s_2$.

The exception table has two important properties that hold during the translation. The first one (Lemma 3) states that the exception entries, whose starting labels appear after the last label generated by the translation, are kept unchanged. The second one (Lemma 4) expresses that the exception entry is not changed by the division. These properties are used to prove soundness of the translation. The proof of these lemmas are presented in Appendix D.

**Lemma 3.**

If

$$\Delta_S \left( \frac{\text{Tree}}{P} \frac{s}{\{ Q_n, Q_b, Q_e \}}, l_a, l_{b+1}, l_{\text{break}}, f, et \right) = [(I_{l_a}...I_{l_b}), et']$$

and $l_{\text{start}} \leq l_a < l_b \leq l_{\text{end}}$ then,

for every $l_s, l_e : \text{Label}$ such that $l_b < l_s < l_e \leq l_{\text{end}}$, and

for every $T : \text{Type}$ such that $T \preceq \text{Throwable} \lor T \equiv \text{any}$, the following holds:

$$et[l_{\text{start}}, l_{\text{end}}, T] = et'[l_s, l_e, T]$$

**Lemma 4.** Let $r : \text{ExcTableEntry}$ and $et' : \text{ExcTable}$ be such that $r \in et'$.

If $et : \text{ExcTable}$ and $l_s, l_e : \text{Label}$ are such that $et = \text{divide}(et', r, l_s, l_e)$, then

$$et[l_s, l_e, T] = r[T]$$

**8.2.6 Soundness Theorem**

The theorem states that if (1) we have a valid source proof for the instruction $s_1$, and (2) we have a translation from the source proof that produces the instructions $I_{l_{\text{start}}}...I_{l_{\text{end}}}$, their respective preconditions $E_{l_{\text{start}}}...E_{l_{\text{end}}}$, and the exception table $et$, and (3) the exceptional postcondition in the source logic implies the precondition at the target label stored in the exception table for all types $T$ such that $T \preceq \text{Throwable} \lor T \equiv \text{any}$ but considering the value stored in the stack of the bytecode, and (4) the normal postcondition in the source logic implies the next precondition of the last generated instruction (if the last generated instruction is the last instruction of the method, we use the normal postcondition in the source logic), (5) the break postcondition implies finally properties. Basically, the finally properties express that for every triple stored in $f$, the triple holds and the break postcondition of the triple implies the break precondition of the next triple. And
the exceptional postcondition implies the precondition at the target label stored in the exception table $et_i$ but considering the value stored in the stack of the bytecode. Then, we have to prove that every bytecode specification holds ($\vdash \{E_i\} I_i$).

In the soundness theorem, we use the following abbreviation: for an exception table $et$, two labels $l_a, l_b$, and a type $T$, $et[l_a, l_b, T]$ returns the target label of the first $et$’s exception entry whose starting and ending labels are less or equal and greater or equal than $l_a$ and $l_b$, respectively, and whose type is a supertype of $T$.

The soundness theorem is defined as follows:

**Theorem 7** (Soundness of the Translation from Java to Java Bytecode).

\[
\vdash \text{Tree} \{ P \} s_1 \{ Q_n, Q_b, Q_e \} \equiv T_{S_1} \land
\]

\[
([I_{\text{start}} \ldots I_{\text{end}}], et) = \nabla s (T_{S_1}, I_{\text{start}}, I_{\text{end}+1}, I_{\text{break}}, f, et') \land
\]

\[
(\forall T : \text{Type} : (T \leq \text{Throwable} \lor T \equiv \text{any}) : \)

\[
(Q_e \land \text{exc} \neq \text{null} \land s(0) = \text{exc} \land \Rightarrow E_{et'[I_{\text{start}}, I_{\text{end}}, T]} \land
\]

\[
(Q_n \Rightarrow E_{l_{\text{end}+1}}) \land
\]

\[
(Q_b \Rightarrow (f = \emptyset \Rightarrow (Q_b \Rightarrow E_{l_{\text{break}}})) \land
\]

\[
(f \neq \emptyset \Rightarrow \forall i \in 1..k : \)

\[
(\vdash \{U_i\} s_i \{ V_i \} \land (V_i \Rightarrow U_i^{i+1}) \land
\]

\[
(\forall T : \text{Type} : (\{T \leq \text{Throwable}\} \lor (T \equiv \text{any}) : \)

\[
(V_i \land \text{exc} \neq \text{null} \land s(0) = \text{exc} \land \Rightarrow E_{et[T]} \land
\]

\[
(Q_b \Rightarrow U_b^1) \land (V_b \Rightarrow E_{l_{\text{break}}}) \land
\]

\[
(\forall et_i = r_i^1 + r_i^2 + \ldots + r_i^m : (r_i^1 \cap r_i^2 \cap \ldots \cap r_i^m = \emptyset) \land (et_i \subseteq et')) \land
\]

\[
\Rightarrow \forall l \in I_{\text{start}} \ldots I_{\text{end}} : \vdash \{E_i\} I_l
\]

The proof runs by induction on the structure of the derivation tree for $\{P\} s_1 \{Q_n, Q_b, Q_e\}$. The proof is presented in Appendix D, Section D.2.

### 8.3 Applications

This section presents an example of the application of the Java proof-transforming compiler. This example uses Java Bytecode and illustrates the translation presented in Section 8.1. The PTC takes the proof described in Section 3.3.4 and produces a bytecode proof. The translation starts with the class rule, then the PTC translates the routine implementation rule, and finally it applies the instruction translation. The bytecode derivation is as follows:
8.3 Applications

{ true }
{s(0) = 1}
{ b = 1 }

// try block
{ b = 1 }
{ b = 1 }

// finally block
{ b = 1 ∧ excV ̸= null ∧ s(0) = excV }
{ b = 1 ∧ eTmp = excV }
{ b = 1 ∧ s(0) = 1 }
{ b = 1 ∧ s(1) = 1 ∧ s(0) = b }
{ b = 1 ∧ s(0) = b + 1 }
{ b = 2 }
{ false }
{ false }

{ b = 1 }
{ b = 1 ∧ s(0) = true }
{ b = 2 }
{ b = 2 ∧ s(0) = 1 }
{ b = 2 ∧ s(1) = 1 ∧ s(0) = b }
{ b = 2 ∧ s(0) = 1 + b }

Exception Table

<table>
<thead>
<tr>
<th>From</th>
<th>to</th>
<th>target type</th>
</tr>
</thead>
<tbody>
<tr>
<td>IL004</td>
<td>IL007</td>
<td>IL007 any</td>
</tr>
</tbody>
</table>

Fig. 8.1: Bytecode Proof generated by the Java PTC from the Example Figure 3.14

{ τ(Current) ≺ MATH ∧ true } MATH:foo { b = 3 , false }
{ τ(Current) = MATH ∧ true } impl(MATH, foo) { b = 3 , false }
{ τ(Current) ⪯ MATH ∧ true } MATH:foo { b = 3 , false }

The first hypothesis:

{ τ(Current) ≺ MATH ∧ true } MATH:foo { b = 3 , false }

is the result of the translation of the false axiom. The second hypothesis is obtained applying the translation of the routine implementation rule. The produced routine implementation rule is the following:
proof Figure 8.1

\[
\begin{array}{l}
\{ \text{true} \} \text{ body(MATH@foo) } \{ b = 3 \ , \ \text{false} \} \\
\{ \text{true} \} \text{ MATH@foo } \{ b = 3 \ , \ \text{false} \}
\end{array}
\]

Figure 8.1 presents the bytecode proof for the body of the routine foo. The try-finally instruction in the source is translated in lines IL004 – IL014. The body of the loop is between lines IL004 and IL016. The break translation yields at line IL012 a br instruction whose target is the end of the loop, i.e., line IL017. Due to the execution of a break instruction, the code from IL017 to IL018 is not reachable (this is the reason for their false precondition).

8.4 Related Work

Proof-Carrying Code and Certifying Compilation

The proof-transforming compilation approach has been inspired by Proof-Carrying Code (PCC) [84, 85, 88]. In PCC, the code producer provides a proof, a certificate that the code does not violate the security properties of the code consumer. Before the code execution, the proof is checked by the code consumer. Only if the proof is correct, the code is executed. With the goal of generating certificates automatically, Necula and Lee [87] have developed certifying compilers. Certifying compilers are compilers that take a program as input and produce bytecode and its proof. Examples of certifying compilers are the Touchstone compiler [85] for a subset of C, and the Special J compiler [29] for a subset of Java.

One of the advantages of Proof-Carrying Code is that mobile code carries a mathematical proof machine checkable. Furthermore, the overhead of producing the proof is done by the code producer. One of the disadvantages of PCC is that the trusted computing base is big, consisting of a verification condition generator, a safety policy, and a proof checker. Furthermore, certifying compilers only work automatically with a restricted set of provable properties such as type safety. Since our approach supports interactive verification of source programs, we can handle more complex properties such as functional correctness. Similar to Foundation Proof-Carrying Code [2], our trusted computing base is reduced to a proof checker.

The open verifier framework for foundational verifiers [27] verifies untrusted code using customized verifiers. The architecture consists of a trusted checker, a fixpoint module, and an untrusted extension (a new verifier developed by untrusted users). The approach is based on Foundation Proof-Carrying Code [2], and it reduces the trusted computing base to the proof checker. However, the properties that can be proved automatically are still limited.
8.4 Related Work

Certified Compilation

Compilers are complex software system. Proving that the implementation of the compiler is correct is hard. A certified compiler [41, 22, 106] is a compiler that generates a proof that the translation from the source program to the assembly code preserves the semantics of the source program. Together with a source proof, this gives an indirect correctness proof for the bytecode program. Leroy [68] defines a certified compiler as a compiler $Comp$ which satisfies the following property:

$$\forall s : \text{Source}, t : \text{Target} : \text{Comp}(s) = t \implies \text{Sem}(s, t)$$

Leroy [68] proves that the semantics of a C program is preserved after compilation to PowerPC assembly code. The innovation of Leroy’s work is that the certified compiler compiles a subset of C commonly used for critical embedded software. The compilation chain is done through four intermediate languages. The compiler has been written in the Coq specification language [19]. The proof is a machine-checked proof, and it has been also formalized in Coq.


Proof-transforming compilers generate the bytecode proof directly, which leads to smaller certificates compared to certified compilers.

Translation Validation

Pnueli et al [104] have developed the concept of translation validation. Instead of proving that the compiler always generates a correct target code, in translation validation each translation is validated showing that the target code correctly implements the source program. The approach compares the input and the output, using an analyzer, independently of how the compiler is implemented. Thus, a change in the compiler does not affect the analyzer. The analyzer takes both the source and target program, and produces a proof script if the target program correctly implements the source program; otherwise, it generates a counter example. Pnueli et al [104] develop a translation from the synchronous multi-clock data-flow language SIGNAL to sequential C code. They generate proof obligations that are solved by a model checker.

Necula [86] implements a translation validation for an optimizing GNU C compiler using symbolic execution. The optimizations include branch optimizations, common subexpression elimination, register allocation, and code scheduling. The experiments developed by Necula reports that the translation validator slows down compilation by a factor of four. Zuck et al. [128] extend the optimizations to structure modifying optimizations such as loop distribution and fusion, loop tiling, and loop interchange.
Tristan and Leroy [121] observes that translation validation provides formal correctness similar to compiler verification if the validator is formally verified. The validator can be formalized as a function \( V: \text{Source} \times \text{Target} \rightarrow \text{Bool} \); then, one has to prove that if \( V(s, t) \) holds, then the source code \( s \) is semantically equivalent to the target code \( t \). They analyze the usability of the verified validator approach for optimizer compilers. The optimizations are list scheduling and trace scheduling (used to improve instruction-level parallelism).

In translation validation, a proof checker can be used to check that the generated proof script is correct. However, the certificate consist of the proof script, the source program, and the target program. To be able to apply translation validation in a similar setting as proof-transforming compilers, one would also need to check that the proof of the source program is valid. Thus, one would need another proof checker, and the certificate would be also bigger.

### Proof-Transforming Compilation

There has been several works on proof-transforming compilation [6, 15, 103, 81, 95]. The closer related work to our proof-transforming compilers are the works by Barthe et al. [15, 9, 10, 13] on proof preserving compilation.

Figure 8.4 shows the general architecture of proof preserving compilation. In the code producer side, the verification condition generator on the source language (\textit{VCGen on Source}) takes the annotated program and its contracts, and generates the verification condition for the source program (\textit{Source VC}). This verification condition is proven using an interactive or automatic prover. The prover generates a certificate which is sent to the code consumer. Together with the certificate, the code consumer receives the bytecode program with annotations and contracts. A verification condition (VC) for the bytecode is generated using the \textit{VCGen on Bytecode}. The VC for the bytecode and the certificate are taken by the proof checker. If the certificate is valid, the bytecode program is executed; otherwise the program is rejected.

Although the certificate was proven using the verification condition generated from the source program, Barthe et al. [15] show that proof obligations are preserved by compilation. Thus, if the certificate proves the source VC, then the certificates also proves the bytecode VC. In an initial work [15], the source language is a subset of Java, which includes method invocation, loops, and exception handling using \textit{throw} and \textit{try-catch} instructions. However, they do not consider \textit{try-finally} instructions, which make the translation significantly more complex. Our translation supports \textit{try-finally} and \textit{break} instructions.

Pavlova et al. [103, 10] extends the aforementioned work to a subset of Java (which includes \textit{try-catch}, \textit{try-finally}, and \textit{return} instructions). They prove equivalence between the VC generated from the source program and the VC generated from the bytecode program. The translation of the above source language has a similar complexity to our Java PTC. However, Pavlova avoided the code duplication for \textit{finally} blocks by disallow-
8.4 Related Work

Fig. 8.2: General Architecture of Proof Preserving Compilation.

...ing return instructions inside the try blocks of try-finally instructions. This simplifies not only the verification condition generator, but also the translation and the soundness proof.

Comparing our Eiffel PTC and Barthe et al.’s work [15, 103, 10], the translation in their case is less difficult because the source and the target languages are closer. However, their work does not address the translation of specifications. Our translation is more complicated due to the translation of contracts.

The Mobius project develops proof-transforming compilers [77, 78]. They translate JML specifications and proof of Java source programs to Java Bytecode. The translation is simpler than our Eiffel PTC because the source and the target language are closer.

In these works on proof preserving compilation [15, 103, 10], the certificate that proves the VC on the source is also used to prove the VC on bytecode. This certificate does not need to be changed because the Java and Java bytecode language are closed. To apply the proof preserving compilation approach to Eiffel, we would need to translate the generated certificate.

[1] The certificate is not changed since they use a bytecode language which has a boolean type, and they assume identifiers for variables are the same as the source language.
In Figure 8.3, we show how to extend the proof preserving compilation approach to Eiffel. In the code producer, the Eiffel program and the contracts are taken by the VCGen on Source, and the VCGen produces the verification condition. This Source VC is proven using an interactive or automatic prover. Since the class structure of Eiffel programs is different to the class structure of CIL programs, the source certificate has to be translated to a CIL certificate. This task is performed by the Certificate Translator. The Eiffel program is compiled to a bytecode program with annotations, and the Eiffel contracts are translated to CIL by the Contract Translator. An example of the definition of the contract translator is the contract translator defined in Section 7.1. Finally, the bytecode VC is generated by the VCGen on Bytecode, and the CIL certificate is checked using the Proof Checker.

The above described works have been applied to non-optimizing compilers. Barthe et al. [9] and Saabas and Uustalu [114] have extended proof-transforming compilation to optimizing compilers. Barthe et al. [9] translate certificates for optimizing compilers from a simple imperative language to an intermediate RTL language (Register Transfer Language). The translation is done in two steps: first the source program is translated into RTL and then optimizations are performed building the appropriate certificate. The source language used in their work is simpler than ours. Saabas and Uustalu [112, 114, 113] describe how optimizations can be applied to proof-transforming compilers. They apply
their technique to a simple while language. We will investigate optimizing compilers for object-oriented languages as part of future work.

Kunz et al. [13] extend the preservation of proof obligations for hybrid verification methods based on static analysis and verification condition generation. They have applied their technique to an imperative language. Barthe and Kunz [12, 61] study certificate translation for program transformation using abstract interpretation [31]. They define the certificate as an abstract notion of proof algebra that includes a set of functions used to define a certificate translator for program transformation. Instead of handling concrete program optimizations, they consider arbitrary program transformations such as code duplication, and subgraph transformation. The same authors [11] study proof compilation for aspect oriented programming.

This chapter extends Müller and Bannwart’s work [6] on proof-transforming compilation. They present a proof-transforming compiler from a subset of Java which includes loops, conditional instructions and object-oriented features. The architecture of Müller and Bannwart’s PTC is the same as our compiler: the PTC takes a proof of the source program (a derivation in a Hoare-style logic), and produces a bytecode proof (a derivation in the bytecode logic). The logic used in Müller and Bannwart’s work is different from the logic used in this chapter. In Chapter 6, our work handles assignment which might trigger exception, while their work does not handle exceptions. Then, we have extended the PTC to Eiffel and Java, which produces a more complex translation. Furthermore, we have formalized and proven soundness.

**Bytecode Verification**

Verification techniques for Bytecote programs have been developed. The approach consists of developing the proof on the bytecode program instead of developing the proof on the source program. An example of bytecode verification is the work developed by the Mobius project [76]; they present a program logic for a bytecode language similar to Java bytecode. The logic has been proved sound with respect the operational semantics, and it has been formalized in Coq. The logic is part of the proof-transforming compilation approach described in the above section.

Liu and Moore [69] have defined a deep embedding of Java Bytecode in ACL2. They show that Java program verification via a deep embedding of the Java bytecode into the logic of ACL2 is a viable approach. They reason about the Java programs by compiling the programs to Java bytecode, and then applying the verification on the compiled code. Our approach verifies the source program, and then it produces the bytecode proof automatically. We think verifying the source program is simpler than verifying the bytecode program.

Other logics for bytecode have been developed such as Quigley’s logic [110] which formalizes Java Bytecode in Isabelle; Dong et al. [37] develop a Hoare-style logic for bytecode; Luo et al. [70] extend separation logic to Java Bytecode. Benton [16] presents
a Hoare-style logic for a stack-based imperative language with unstructured control ow
similar to .NET CIL. However, the logic does not support object-oriented features such
as inheritance. Bannwart and Müller [6] have developed a Hoare-style logic for a bytecode
similar to Java Bytecode and CIL. The logic supports object-oriented features such as
inheritance and dynamic binding. However, the bytecode logic does not include exception
handling; so it does not include the CIL instructions .try catch and .try .finally.
To show the feasibility of proof-transforming compilers, we have implemented a proof-transforming schema for a subset of Eiffel. Figure 9.1 shows the architecture of the implementation. An Eiffel program is verified using a Prover (the implementation of the Prover is out of the scope of this thesis). The generated proof is encoded in an XML format. The proof-transforming compiler takes as input the source proof encoded in XML, and produces as output a CIL proof. The CIL proof is embedded in an Isabelle theory. The proof checker, which has been implemented in Isabelle, takes this embedded proof, and checks if the proof is valid or not. If the proof is valid, the program can be executed, otherwise the program is rejected.

In the following sections we describe the implementation of the proof-transforming compiler and the formalization of proof checker in Isabelle. More details can be found in the ETH reports.

9.1 A Proof-Transforming Compiler

The proof-transforming compiler takes the Eiffel proof in a XML file, and produces an Isabelle theory. This PTC consists of an XML Parser, a Proof Translator, a Contract Translator, and an Isabelle Generator. The architecture of the Eiffel PTC compiler is presented in Figure 9.2.

The XML Parser takes the Eiffel proof, and generates an abstract syntax tree (AST). This parser not only generates an AST for the Hoare triples but also for the pre and postconditions used in these triples. Furthermore, it generates an AST for the contracts.
These ASTs are needed to be able to translate contracts, preconditions, and postcondition to CIL. Also, the ASTs are used to embed the proof into Isabelle.

After parsing the proof, the Contract Translator visits the generated AST, and translates to CIL the contracts, and pre and postconditions of the proof. The Contract Translator implements the translation functions described in Section 7.1. This translation does not generate the CIL proof jet, it only maps Eiffel expressions to CIL.

Once the Eiffel contracts, and pre and postconditions have been translated to CIL, the Proof Translator generates the CIL proof. The Proof Translator produces a representation of the CIL proof in an AST. This module is implemented visiting the AST for the Eiffel proof, and generating the AST for the CIL proof. The complexity of the implementation of the Proof Translator is similar to a standard compiler.

One of the most interesting part in the implementation of the Proof Translator is the translation of weak and strength rules. The application of these rules contains a proof of the form $P' \Rightarrow P$. To be able to translate these proofs, we assume that the proofs have been developed in Isabelle. The XML file has the Isabelle proof that shows $P' \Rightarrow P$. The Proof Translator generates a lemma $Q' \Rightarrow Q$ where $Q'$ and $Q$ are the translation of the
expressions $P'$ and $P$ resp. into Isabelle ($Q' = \nabla_C(P')$ and $Q = \nabla_C(P)$). To prove this lemma, we use the proof provided in the XML file.

Finally, the Isabelle Generator produces an Isabelle [90] theory from the AST of the bytecode proof. This module embeds CIL contracts, preconditions, and bytecode instructions into Isabelle. Furthermore, it generates a proof script that shows that the bytecode proof is valid. This proof script, first, unfolds the class, method, instruction and precondition definitions. Second, it applies the definition of $wp$, $shift$, and $unshift$, and finally it simplifies the expressions.

### 9.2 Proof Checker

We have implemented a proof checker for a subset of the CIL logic. The subset includes the rules for the main CIL instructions. However, it omits the rules for method specifications (such as the routine implementation rule, the class rule, and the subtype rule), and the language-independent rules (applied to method specifications).

The proof checker takes a list of instruction specifications. The rules for the CIL instruction specifications have the form:

$$E_i \Rightarrow wp(I_i)$$
$$\mathcal{A} \vdash \{E_i\} l : I_i$$

To check the proof, the proof checker verifies that the implications $E_i \Rightarrow wp(I_i)$ hold for all instruction specifications. We have implemented the proof checker as a verification condition generator (VCGen). Given a CIL proof embedded in Isabelle, the VCGen gen-
erates verification conditions (VC) of the form $E_l \Rightarrow \wp(I_l)$. If these VCs are true, then the CIL proof is valid. To define the $VCGen$, we first introduce the data type definitions for CIL instructions, exception tables, and CIL proofs. Then, we define the $VCGen$ using the weakest precondition function (defined in Section 5.2.2).

Following, we present the core of the proof checker, for a complete definition formalized in Isabelle see [98].

9.2.1 Instructions

The following data type defines CIL instructions. This definition includes load and store instructions, binary operations such as $\text{add}$ and $\text{ceq}$, branch instructions, load and store fields, object creation, method invocation, and throw instruction. Variables, field names, and method names are denoted by $\text{VarId}$, $\text{FldId}$, and $\text{MethodId}$ respectively; while values, labels, and types are denoted by $\text{Value}$, $\text{Label}$, and $\text{Type}$ respectively. The definition is:

```
datatype Instruction = ldloc VarId
                     | ldc Value
                     | stloc VarId
                     | add | sub
                     | mul | div
                     | cneq | ceq
                     | cgt | clt
                     | cgte | clte
                     | and | or
                     | neg
                     | br Label
                     | brtrue Label
                     | brfalse Label
                     | nop
                     | ret
                     | ldfld FldId
                     | stfld FldId
                     | newobj Type
                     | castc Type
                     | callvirt Type MethodId Value
                     | throw
```

9.2.2 Exception Tables

Exception tables are formalized as a list of exception entries ($\text{ExcEntry}$). An exception entry is defined as a starting label $l_s$, an ending label $l_e$, a target label $l_t$, and a type $T$. 
The definition is:

```
datatype ExcEntry = Label × Label × Label × Type
datatype ExcTable = list_of ExcEntry
```

The function is_handled is used to query if an exception is caught by an exception entry or not. Given an exception entry \([l_s, l_e, l_t, T]\), a label \(l\), and a type \(T'\), this function yields true if only if \(T'\) is a subtype of \(T\), and \(l_s \leq l\) and \(l < l_e\). The function handlerEntry, given an exception entry, returns the target label. The definitions of these functions are:

```
is_handled : ExcEntry × Label × Type → bool
is_handled (l_s, l_e, l_t, T) l T'  = if (l_s ≤ l ∧ l < l_e ∧ T' ⪯ T) then true
                             else false

handlerEntry : ExcEntry → Label
handlerEntry (l_s, l_e, l_t, T) = l_t
```

Given an exception table \(et\), a label \(l\), and a type \(T\), the function is_caught returns true if only if the exception \(T\) at label \(l\) is caught by the exception table \(et\). The definition is:

```
is_caught : ExcTable × Label × Type → bool
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)
```

The function handler, given an exception table, a label \(l\), and a type \(T\), returns the target label where the exception of type \(T\) is caught (if the exception table does not catch the exception \(T\) at label \(l\), this function returns an arbitrary label). This function is defined as follows:

```
handler : ExcTable × Label × Type → Label
handler [] l T  = arbitrary
handler (x#xs) l T  = if (is_handled x l T) then (handlerEntry x)
                                   else (handler xs l T)
```

### 9.2.3 CIL Proofs

In the implementation of the proof checker, CIL proofs are defined as a list of class declarations. A class declaration consists of a type and a class body. The class body is defined as a list of method declarations. This method declaration is implemented as a method id,
Implementation of the Proof-Transforming Compiler Scheme

a precondition, a CIL method body proof, an exception table, and a postcondition. To simplify the proof checker, we have not implemented the rules that are applied to virtual routines or routine implementations (such as the class rule). Furthermore, we omitted the language-independent rules. Thus, method body proofs are a list of instruction specifications. Extending the proof checker is part of future work.

Preconditions and postconditions are boolean expressions. Postconditions are a tuple of boolean expressions: the first element stores the normal postcondition, and the second one the exceptional postcondition. The definition of CIL proofs is the following:

```plaintext
datatype BoolExp = Bool
datatype Precondition = BoolExp
datatype Postcondition = BoolExp BoolExp
datatype InstSpec = BoolExp Label Instruction
datatype CilBodyProof = list of InstSpec
datatype MethodDecl = MethodID Precondition CilBodyProof ExcTable Postcondition
datatype ClassBody = list of MethodDecl
datatype ClassDecl = Type ClassBody
datatype CilProof = list of ClassDecl
```

### 9.2.4 Weakest Precondition Function

To be able to define the proof checker, we have formalized in Isabelle the definition of the weakest precondition function presented in Section 5.2.2. For simplicity, we omit the load and store field instructions as well as the newobj instruction. The complete definition can be found in [98].

This formalization uses the substitution functions subst\(_x\), subst\(_c\), subst\(_c1\), and subst\(_\sigma\). The function subst\(_x\) substitutes s(0) by a variable name. The function subst\(_c\) substitutes s(0) by a constant c, and subst\(_c1\) substitutes s(1) by a constant c. The function subst\(_\sigma\) substitutes a variable name in state \(\sigma\) by s(0). The definition of \(Wp\) is the following:
9.2 Proof Checker

Wp : Instruction \times \text{BoolExp} \times \text{CilProof} \times \text{BoolExp} \rightarrow \text{BoolExp}

\begin{align*}
Wp \ \text{inst sucPre proof post} &= \text{case inst of} \\
\text{lloc } v &\rightarrow (\text{unshift}(\text{subst}_x \text{sucPre } v)) \\
\text{ldc } n &\rightarrow (\text{unshift}(\text{subst}_c \text{sucPre } n)) \\
\text{stloc } v &\rightarrow (\lambda s: \text{Stack } \sigma : \text{State} \ (\text{subst}_\sigma (\text{shift sucPre}) v)) \\
\text{binop} &\rightarrow (\lambda s: \text{Stack } \sigma : \text{State} \ (\text{subst}_{c1} (\text{shift sucPre}) (s(1) \text{ binop } s(0)))) \\
\text{unop} &\rightarrow (\lambda s: \text{Stack } \sigma : \text{State} \ (\text{subst}_{c1} (\text{shift sucPre}) (\text{unnop } s(0)))) \\
\text{br } l_2 &\rightarrow (\text{PrecLabel proof } l_2) \\
\text{brtrue } l_2 &\rightarrow (\lambda s : \text{Stack } \sigma : \text{State} \ (\neg s(0) \Rightarrow (\text{shift sucPre})) \land \\
&s(0) \Rightarrow (\text{shift(PrecLabel proof } l_2))) \\
\text{brfalse } l_2 &\rightarrow (\lambda s : \text{Stack } \sigma : \text{State} \ (s(0) \Rightarrow (\text{shift sucPre})) \land \\
&\neg s(0) \Rightarrow (\text{shift(PrecLabel proof } l_2))) \\
\text{nop} &\rightarrow \text{sucPre} \\
\text{ret} &\rightarrow \text{post}
\end{align*}

where \text{PrecLabel} (\text{proof } l) returns the precondition at label \( l \).

9.2.5 Verification Condition Generator

The verification condition generator (\text{VCGen}) produces a proof obligation (a list of verification conditions). The verification condition (\text{VC}), and the proof obligation data types are defined as follows:

\begin{align*}
\text{datatype VC} &= \text{bool} \\
\text{datatype ProofObligation} &= \text{list_of VC}
\end{align*}

The verification condition generator is defined by the functions \text{VCGenInst}, \text{VCGenMethod}, \text{VCGenClassBody}, \text{VCGen2}, and \text{VCGen}. The function \text{VCGenInst} generates verification conditions for instruction specifications. Except for \text{callvirt} and \text{throw}, this function returns the implication of the precondition of the instruction specification and the weakest precondition of the next instruction. To obtain the weakest precondition and the precondition of the next instruction, this function takes a CIL body proof, and a CIL proof. The proof obligations for \text{callvirt} and \text{throw} are generated by the functions \text{proof_oblig_call} and \text{proof_oblig_throw} respectively. The definition is:
VCGenInst : InstSpec × ExcTable × CilBodyProof × CilProof × BoolExp × BoolExp → VC
VCGenInst (pre, l, inst) et cilProof prog postN postE = case i of
  callvirt T m e → (proof_oblig_call T m e pre l cilProof postN)
  | throw → (pre ⇒ (proof_oblig_throw et l postE cilProof))
  | _ → (pre ⇒ (Wp inst l cilProof postN))

The function proof_oblig_call returns a boolean expression that express that (1) the precondition of the instruction specification implies the precondition of the method where the formal argument and this are replaced by s(0) and s(1) respectively, and (2) the postcondition of the method, where the formal argument is replaced by e, and s(0) is replaced by Result, implies the postcondition of the instruction specification. The definition is:

proof_oblig_call : CilProof × Type × MethodID × Value × BoolExp × BoolExp → bool
proof_oblig_call prog T m e pre post =
  (pre ⇒ (λ s : Stack σ : State.
            substσ₁(substσ(get_prec prog T m) PARAM ) THIS)) ∧
  (substσ₂(substpv(get_post prog T m) e) RESULT) ⇒ post)

where the function substσ substitutes a variable name in state σ by s(0), the function substσ₁ substitutes a variable name in state σ by s(1), substpv substitutes the expression e by p, and get_prec and get_post returns the precondition and postcondition of a method m respectively.

The function proof_oblig_throw defines the proof obligation for the throw instruction. This function uses the function handlerPost, which returns the precondition of the instruction where control is transferred after an exception of type T is triggered at label l. The definition is:

handlerPost : ExcTable × Label × Type × BoolExp × CilProof → BoolExp
handlerPost et l T post proof = if (is_caught et l T) then
  (PrecLabel proof (handler et l T))
else post
where PrecLabel (proof l) returns the precondition at label l

proof_oblig_throw : ExcTable × Label × BoolExp × CilProof → BoolExp
proof_oblig_throw et l post proof = (λs.(handlerPost et l (τ s(0))) post proof)

The function VCGenMethod produces proof obligations for a method body proof. This function invokes VCGenInst per every instruction specification. The definition is:
The function \( VCGenClassBody \) generates proof obligations for a class body, and the functions \( VCGen2 \) and \( VCGen \) produces proof obligations for CIL proofs. These functions are defined as follows:

\[
\begin{align*}
VCGenClassBody : & \text{ClassBody} \times \text{CilProof} \rightarrow \text{ProofObligation} \\
VCGenClassBody \; & [\;] \; \text{prog} \; = \; [\text{true}] \\
VCGenClassBody \; & (x \#\text{xs}) \; \text{prog} \; = \\
& \quad (\text{case } x \; \text{of } (m, \text{pre}, \text{cilP}, \text{et}, \text{post}) \rightarrow \\
& \quad \quad (\text{case } \text{post} \; \text{of } (\text{postN}, \text{postE}) \rightarrow \\
& \quad \quad \quad (VCGenMethod \; \text{cilP} \; \text{et} \; \text{cilP} \; \text{prog} \; \text{postN} \; \text{postE}) \; + \\
& \quad \quad \quad (VCGenClassBody \; \text{xs} \; \text{et} \; \text{cilP} \; \text{prog} \; \text{postN} \; \text{postE}))
\end{align*}
\]

\[
\begin{align*}
VCGen2 : & \text{CilProof} \times \text{CilProof} \rightarrow \text{ProofObligation} \\
VCGen2 \; & [\;] \; \text{prog} \; = \; [\text{true}] \\
VCGen2 \; & ((\text{type}, \text{body}) \#\text{xs}) \; \text{prog} \; = \; (VCGenClassBody \; \text{body} \; \text{prog}) \; + \; (VCGen2 \; \text{xs} \; \text{prog})
\end{align*}
\]

\[
\begin{align*}
VCGen : & \text{CilProof} \rightarrow \text{ProofObligation} \\
VCGen \; & c \; = \; VCGen2 \; c \; c
\end{align*}
\]

## 9.2.6 Checking the CIL Proofs

To verify that a CIL proof is valid, the proof checker, first, generates the proof obligations, and then it tries to prove that these proof obligations are true. The PTC produces a proof script that shows that the proof obligations holds. Using this proof script, the proof checker can check the proof automatically.

To check that a bytecode proof is valid, we proof the following theorem:

**Theorem 8** (Proof Checker). Given a CIL proof \( p : \text{CilProof} \), then

\[
is\_\text{program}\_\text{safe}(VCGen \; p) \; \text{holds}.
\]

The function \( is\_\text{program}\_\text{safe} \) is used to prove that a proof obligation is valid. Given a proof obligation (a list of verification conditions), this function checks that all verification conditions are true. The function is defined as follows:
Implementation of the Proof-Transforming Compiler Scheme

\[ \text{is\_program\_safe} : \text{ProofObligation} \rightarrow \text{bool} \]

\[
\begin{align*}
\text{is\_program\_safe \;[]} & = \text{true} \\
\text{is\_program\_safe \;(x \# xs)} & = \text{if } x \text{ then } (\text{is\_program\_safe \;xs}) \text{ else } \text{false}
\end{align*}
\]

9.3 Experiments

Using the implemented prototype, we have been able to perform some experiments. These experiments show the size of the source proof, the size of the generated certificate, and the time expended to check the certificate. Although the examples are small, they suggest that proof transforming compilation can be applied to object-oriented programs. As future work, we plan to develop a bigger case study to be able to conclude whether the technique can be applied to real programs or not.

Table 9.1 shows the size of the source program, the size of the source proofs, and the size of the certificates. The first four examples are simple example involving the use of boolean expressions, attributes, conditionals, and loops respectively. The fifth example is a typical bank account example with routines for withdraw and deposit. The Sum Integer example takes an integer \( n \) and returns the sum of 1 to \( n \); the subtype example implements a class person with two descendants classes: students and teacher. The last example contains all the previous examples and some other basic instructions. In average, the size of the certificates are three times the size of the source proof.

<table>
<thead>
<tr>
<th>Example</th>
<th>#Classes</th>
<th>#Routines</th>
<th>#lines in Eiffel</th>
<th>#lines source proof</th>
<th>#lines in Isabelle</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Boolean exprs</td>
<td>1</td>
<td>3</td>
<td>50</td>
<td>111</td>
<td>459</td>
</tr>
<tr>
<td>2. Attributes</td>
<td>2</td>
<td>3</td>
<td>83</td>
<td>167</td>
<td>1141</td>
</tr>
<tr>
<td>3. Conditionals</td>
<td>1</td>
<td>2</td>
<td>55</td>
<td>154</td>
<td>510</td>
</tr>
<tr>
<td>4. Loops</td>
<td>1</td>
<td>1</td>
<td>31</td>
<td>73</td>
<td>305</td>
</tr>
<tr>
<td>5. Bank Account</td>
<td>1</td>
<td>5</td>
<td>57</td>
<td>130</td>
<td>596</td>
</tr>
<tr>
<td>6. Sum Integers</td>
<td>1</td>
<td>1</td>
<td>35</td>
<td>126</td>
<td>358</td>
</tr>
<tr>
<td>7. Subtyping</td>
<td>3</td>
<td>5</td>
<td>41</td>
<td>117</td>
<td>756</td>
</tr>
<tr>
<td>8. Demo</td>
<td>18</td>
<td>28</td>
<td>548</td>
<td>1306</td>
<td>3718</td>
</tr>
</tbody>
</table>

Tab. 9.1: Experiments with the implemented PTC scheme: lines of code of the source program, the source proof, and the produced certificate.

Table 9.2 presents the time expended to check the bytecode proofs. The experiments were run on a machine with a 2.5 GHz dual core Pentium processors with 2GB of RAM. We have developed two generators for proof script. The first one adds all the definitions to the simplifier and then invokes the tactic \texttt{apply simp}. The second one generates a proof script that first unfolds the definition of the class proofs, routine proofs, and instructions
specification proof; and then it applies the definition of the weakest precondition function, and finally invokes the tactic \textit{apply simp}. The results show that the optimized script reduces the time expended to the half. However, we believe that the proof script can be optimized to reduce the checking time.

<table>
<thead>
<tr>
<th>Isabelle Example</th>
<th>#lines in Isabelle</th>
<th>Simplifier Proof Script</th>
<th>Optimized Proof Script</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Boolean expressions</td>
<td>459</td>
<td>9.4 sec.</td>
<td>6.9 sec.</td>
</tr>
<tr>
<td>2. Attributes</td>
<td>1141</td>
<td>7.6 sec.</td>
<td>6.2 sec.</td>
</tr>
<tr>
<td>3. Conditionals</td>
<td>510</td>
<td>9.3 sec.</td>
<td>7.8 sec.</td>
</tr>
<tr>
<td>4. Loops</td>
<td>305</td>
<td>18.1 sec.</td>
<td>7.2 sec.</td>
</tr>
<tr>
<td>5. Bank Account</td>
<td>596</td>
<td>16.8 sec.</td>
<td>8.6 sec.</td>
</tr>
<tr>
<td>6. Sum Integers</td>
<td>358</td>
<td>45.2 sec.</td>
<td>6.3 sec.</td>
</tr>
<tr>
<td>7. Subtyping</td>
<td>756</td>
<td>8.3 sec.</td>
<td>4.3 sec.</td>
</tr>
<tr>
<td>8. Demo</td>
<td>3718</td>
<td>5 min.</td>
<td>2 min.</td>
</tr>
</tbody>
</table>

Tab. 9.2: Experiments with the implemented PTC scheme: time expended to check the proof.
Implementation of the Proof-Transforming Compiler Scheme
Chapter 10

Conclusions and Future Work

10.1 Conclusions

We have presented an approach to verifying mobile code. The approach consists of developing a proof for the source program, and then translating it automatically to bytecode using a proof-transforming compiler (PTC). This thesis has shown that it is feasible to generate bytecode proofs from proofs for object-oriented programs. The approach is heterogeneous on the programming language: programs verified in multiple languages such as C#, Eiffel, and Java can be translated to a common bytecode logic.

We have developed proof transformation for a wide range of features of object-oriented programs such as different exception handling mechanisms, single and multiple inheritance, and once routines. We have defined proof-transforming compilers for C#, Eiffel, and Java. The formalization of the Eiffel PTC has risen challenges because of the difference between the source language, Eiffel, and the target language, CIL. The Java PTC has shown the difficulties of formalizing proof transformation for try-finally and break instruction. Furthermore, we have formalized the translation of a subset of Java using two different bytecode languages: CIL and JVM. These PTCs have shown that the formalization of a subset of Java with try-finally and break instruction is more complex than formalizing only try-finally instructions.

To develop the proof of the source program, we have developed semantics for C#, Eiffel, and Java. The main new features that are included in the semantics are the treatment of exception handling. The subset of Java includes abrupt termination using the break, throw, try-catch, and try-finally instructions. The subset of Eiffel handles features such as exception handling (through rescue clauses), multiple inheritance, and once routines. The semantics have been formalized using operational and axiomatic semantics. We have proven soundness and completeness of the semantics.

Furthermore, we have presented a specification and verification methodology for function objects using Eiffel agents. We have shown the application of the agent methodology
through a set of non-trivial examples, and we have implemented a prototype, which automatically proves these examples.

Proof-transforming compilers have been formalized not only for a subset of Eiffel, but also for a subset of Java. In the case of the Eiffel proof-transforming compiler, the semantics difference between Eiffel and CIL made the translation harder. The problems were produced by the lack of multiple inheritance in CIL. Besides the translation of proofs for Eiffel programs, the Eiffel PTC includes a contract translator to map Eiffel contracts to CIL contracts. Although we have use CIL bytecode, the Eiffel translation can be adapted to other bytecode languages such as JVM.

The proof-transforming compiler for Java has been developed for both CIL and JVM bytecode. We have shown that the translation using JVM is more complex due to the generation of exception tables. These complications were produced by the compilation of `break` and `try-finally` instructions.

The trusted computing base (TCB) of our approach consist only of a proof checker. Although proof-transforming compilers are not part of the TCB, a soundness result is interesting from the point of view of the code producer. We have formalized and proven soundness for the developed proof-transforming compilers.

To show a feasibility of the use of PTCs, we have implemented a PTC for Eiffel and a proof checker. The compiler automatically generates a bytecode proof, and a proof script that shows that the bytecode proof is valid. This proof script is checked (automatically) by the proof checker. The proof checker has been formalized in Isabelle.

10.2 Future Work

10.2.1 Proofs

Semantics. We have formalized a semantics for C#, Eiffel, and Java. In the case of Java, our formalization of `try-finally` instructions has clarified its semantics. In the case of Eiffel, the formalization of exception handling has introduced interesting discussions, and it has introduced changes in the language. To understand better the programming languages, we would need to formalize the full language. Although previous works have formalized subsets of C#, Eiffel and Java, there is not any work that formalizes the semantics of the whole language. For example, the Eiffel ECMA standard [75] has contributed to improve the understanding of the language. However, new concepts such as `object test`, and `attached types` need to be formalized.

Specification Languages. In Chapter 4 we have proposed a specification and verification methodology for agents. To specify routines that apply an agent to a sequence, such as the routines `do_all` and `do_if` in the class `LINEAR`, we use a non-interference operator. In practice, however, a sequence could have repeated elements; thus one could not invoke
routines such as `do_all`, and `do_if`. Specification languages need to be extended to able to specify more complex programs using agents.

**Tools.** One of the goals of the verification effort is to support automatic verification of object-oriented programs. Tools such as Spec# [8] have made a good progress to achieve this goal, however, there is still a long way to go until these tools are applied to real object-oriented programs. Tools should be easy to use so that programmers can use them, but also they should be powerful enough to be able to prove interesting programs.

Case studies have to be applied to real applications. These case studies will be useful to discover limitations and to develop new approaches.

### 10.2.2 Proof-Transforming Compilation

**Translations.** The proof transformations presented in this thesis handles a subset of C#, Eiffel, and Java. One direction for future work is to extend the translation of features such as agents in Eiffel, and delegates in C#.

Another direction for future work is to study how to reduce the size of the certificates. The certificates produced by our proof transforming compiler contains the precondition of each instruction specification. In Barthe et al.’s work [15, 10, 12], these intermediate steps are not part of the certificate. An interesting case of study is to analyze in depth the different approaches comparing size of the certificate, and speed of the proof checker. This study may conclude that some intermediate steps could be eliminated since the proof checker could generate them.

Several assumptions about the bytecode language have been applied to simplify the translation. We assume that the bytecode language has a boolean type, and we assumed identifiers for variables and fields are the same as the source language. CIL and JVM do not have a boolean type; booleans are compiled to integers. Furthermore, local variables are encoded as elements of a register table. Thus, to apply the proof translation to a real bytecode language, one would need to extend the presented translations. Although the extension is simple, the changes would affect not only the translation of instructions but also the translation of pre- and postconditions in the source proof, as well as the translation of contracts.

**Optimizations.** To be able to develop PTCs for real programs, we need to extend the translation functions described in this thesis to optimizing compilers. Proof-transforming compilers for optimizing compilers have been studied by Barthe et al [9] and Saabas and Uustalu [114]. However, we would need to investigate how to apply the optimizations to our settings.
Tools. An important part of future work is the implementation of the proof-transforming compilation schema. Although we have implemented a prototype, a tool that can translate a substantial subset of Eiffel or Java is still missing. This tool should be integrated as part of the verification tool mentioned in the above section. Thus, the programmer could write the program in the tool and automatically generate the bytecode proof. Furthermore, case studies have to be developed to analyze the feasibility of the use of the approach in real applications.

The only trusted component of our approach is the proof checker. Another direction for future work is to prove that the proof checker is correct.
## Appendix A

### Notation

The following table presents the naming conventions used in this thesis.

Tab. A.1: Naming conventions

<table>
<thead>
<tr>
<th>Type</th>
<th>Typical use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td>$s, s_1, s_2$</td>
</tr>
<tr>
<td>Expressions</td>
<td>$e, e_1, e_2$</td>
</tr>
<tr>
<td>Virtual Routines</td>
<td>$T:m$, $S:m$</td>
</tr>
<tr>
<td>Routine implementations</td>
<td>$T@m, S@m$</td>
</tr>
<tr>
<td>Attributes (fields in Java)</td>
<td>$T@t$</td>
</tr>
<tr>
<td>Types</td>
<td>$T, T', T_1, \ldots, T_n, S$</td>
</tr>
<tr>
<td>States</td>
<td>$\sigma, \sigma', \sigma'', \sigma'''$</td>
</tr>
<tr>
<td>Object Store</td>
<td>$$, $$'$</td>
</tr>
<tr>
<td>Precondition</td>
<td>$P, P', P'', P'''$</td>
</tr>
<tr>
<td>Normal Postcondition</td>
<td>$Q_n, Q'_n, R_n$</td>
</tr>
<tr>
<td>Postcondition for Exceptions</td>
<td>$Q_e, Q'_e, Q''_e, R_e$</td>
</tr>
<tr>
<td>Postcondition for Break</td>
<td>$Q_b, R_b, R'_b$</td>
</tr>
<tr>
<td>Loop Invariant</td>
<td>$I$</td>
</tr>
<tr>
<td>Sequent</td>
<td>$\mathcal{A}, \mathcal{A}_0$</td>
</tr>
<tr>
<td>Label (for examples bytecode)</td>
<td>$IL001, \ldots, IL026$</td>
</tr>
<tr>
<td>Label (for the translation only)</td>
<td>$l_{\text{start}}, l_{\text{exc}}, l_{\text{next}}, l_{\text{break}}, l_{\text{end}}, l_{\text{end}+1}, l_a, l_b, \ldots, l_g$</td>
</tr>
</tbody>
</table>

Continued on next page
Continued from previous page

<table>
<thead>
<tr>
<th>Type</th>
<th>Typical use</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>ProofTree</em> (for source language only)</td>
<td>$T_{S_1}, T_{S_2}, Tree_i, T_{sm}, T_{imp}, T_{tm}$, $T_{body}, T_A, T_{A_0}$</td>
</tr>
<tr>
<td>Bytecode Instruction at label $l$</td>
<td>$I_l$</td>
</tr>
<tr>
<td>Precondition at Label $l$ (for Bytecode Logic)</td>
<td>$E_l$</td>
</tr>
<tr>
<td><em>ProofTree</em> (for source language only)</td>
<td>$T_{S_1}, T_{S_2}, Tree_i$</td>
</tr>
<tr>
<td><em>ProofTree</em> (for <strong>finally</strong> only)</td>
<td>$T_{F_i}$</td>
</tr>
<tr>
<td>List[Finally]</td>
<td>$f, f_i$</td>
</tr>
<tr>
<td><em>ExceptionTable</em></td>
<td>$e_{t_i}$</td>
</tr>
<tr>
<td><em>ExceptionTable</em> (for <strong>finally</strong> only)</td>
<td>$e_{t_i}'$</td>
</tr>
<tr>
<td><em>BytecodeProof</em></td>
<td>$B_{S_1}, B_{S_2}$</td>
</tr>
<tr>
<td><em>InstrSpec</em></td>
<td>$b_{pushc}, \ldots, b_{brtrue}$</td>
</tr>
<tr>
<td><em>Label</em></td>
<td>$l_{start},$</td>
</tr>
<tr>
<td></td>
<td>$l_b, l_c, \ldots, l_g$</td>
</tr>
</tbody>
</table>
This Appendix presents the soundness and completeness proof of the logic for Eiffel. This proof includes the soundness and completeness proof of the logic for the Mate language.

To handle recursive calls, we define a richer semantic relation $\rightarrow_N$ where $N$ captures the maximal depth of nested routine calls which is allowed during the execution of the instruction. The transition $\sigma, S \rightarrow_N \sigma'$, normal expresses that executing the instruction $S$ in the state $\sigma$ does not lead to more than $N$ nested calls, and terminates normally in the state $\sigma'$. The transition $\sigma, S \rightarrow_N \sigma'$, exc expresses that executing the instruction $S$ in the state $\sigma$ does not lead to more than $N$ nested calls, and terminates with an exception in the state $\sigma'$.

The rules defining $\rightarrow_N$ are similar to the rule of $\rightarrow$ presented in Section 3.1.3 and Section 3.2.2 except for the additional parameter $N$. For the rules that do not describe the semantics of neither a routine call, nor a once routine, nor a creation procedure, we replace $\rightarrow$ by $\rightarrow_N$. For example, the compound rule (3.2.4) (described on page 23) is defined as follows:

\[
\langle \sigma, s_1 \rangle \rightarrow_N \sigma', \text{normal} \quad \langle \sigma', s_2 \rangle \rightarrow_N \sigma'', \chi \\
\langle \sigma, s_1; s_2 \rangle \rightarrow_N \sigma'', \chi
\]

**Routine Invocations.** The routine invocation rule described in Section 3.1.3 is extended using the transition $\rightarrow_N$ as follows:

\[
T : m \text{ is not a once routine} \\
\sigma(y) \neq \text{voidV} \\
\langle \sigma[\text{Current} := \sigma(y), p := \sigma(e)], \text{body(impl(\tau(\sigma(y)), m))} \rangle \rightarrow_N \sigma', \chi \\
\langle \sigma, x := y, T : m(e) \rangle \rightarrow_{N+1} \sigma'[x := \sigma'(\text{Result})], \chi
\]

(B.1)
Once Routines. The only rules of once routines that are extended using the transition \( \rightarrow_N \) are the rules that describe the execution of the first invocation. These rules are extended as follows:

\[
T'@m = \text{impl}(\tau(\sigma(y)), m) \quad T'@m \text{ is a once routine}
\]

\[
\sigma(T'@m\_done) = \text{false}
\]

\[
\langle \sigma[T'@m\_done := \text{true}, \text{Current} := y, p := \sigma(\epsilon)], \text{body}(T'@m) \rangle \rightarrow_N \sigma', \text{normal}
\]

\[
\langle \sigma, x := y.S:m(e) \rangle \rightarrow_{N+1} \sigma'[x := \sigma'(\text{Result})], \text{normal}
\]

\[
T@m = \text{impl}(\tau(\sigma(y)), m) \quad T@m \text{ is a once routine}
\]

\[
\sigma(T@m\_done) = \text{false}
\]

\[
\langle \sigma[T@m\_done := \text{true}, \text{Current} := y, p := \sigma(\epsilon)], \text{body}(T@m) \rangle \rightarrow_N \sigma', \text{exc}
\]

\[
\langle \sigma, x := y.S:m(e) \rangle \rightarrow_{N+1} \sigma'[T@m\_exc := \text{true}], \text{exc}
\]

This appendix is based on the technical report [94].

B.1 Definitions and Theorems

To handle recursion, in the following we extend the semantics of Hoare triples [48, 50, 49], and the soundness and completeness theorems.

**Definition 4** (Triple |\(\rightarrow\)|). \(\models \{ P \} \quad s \quad \{ Q_n, Q_e \} \) is defined as follows:

- **If** \( s \) **is an instruction, then**
  \(\models \{ P \} \quad s \quad \{ Q_n, Q_e \} \) **if only if:**
  * for all \( \sigma \models P : \langle \sigma, s \rangle \rightarrow \sigma', \chi \) **then**
    - \( \chi = \text{normal} \Rightarrow \sigma' \models Q_n, \) and
    - \( \chi = \text{exc} \Rightarrow \sigma' \models Q_e \)

- **If** \( s \) **is the routine implementation** \( T@m \), **then**
  for all \( \sigma \models P : \langle \sigma, \text{body}(T@m) \rangle \rightarrow \sigma', \chi \) **then**
  - \( \chi = \text{normal} \Rightarrow \sigma' \models Q_n, \) and
  - \( \chi = \text{exc} \Rightarrow \sigma' \models Q_e \)

- **If** \( s \) **is the virtual routine** \( T:m \), **then for all receivers** \( x \) **such that** \( \tau(x) = T \),
  for all \( \sigma \models P : \langle \sigma, \text{body(\text{imp}(\tau(\sigma(x)), m)))} \rangle \rightarrow \sigma', \chi \) **then**
  - \( \chi = \text{normal} \Rightarrow \sigma' \models Q_n, \) and
The definition of $\models \{ P \} \; s \; \{ Q_n, Q_e \}$ (Definition 4) uses the transition $\rightarrow_N$. To handle recursive routine calls, we extend this definition using the transition $\rightarrow_{N+1}$ as follows:

**Definition 5 (Triple $\models_{N}$.)** $\models_{N} \{ P \} \; s \; \{ Q_n, Q_e \}$ is defined as follows:

- If $s$ is an instruction, then $\models_{N} \{ P \} \; s \; \{ Q_n, Q_e \}$ if only if:
  - $\chi = \text{normal} \Rightarrow \sigma' \models Q_n$, and
  - $\chi = \text{exc} \Rightarrow \sigma' \models Q_e$

- If $s$ is the routine implementation $T@m$, then $\models_{N} \{ P \} \; T@m \; \{ Q_n, Q_e \}$ always holds; and $\models_{N+1} \{ P \} \; T@m \; \{ Q_n, Q_e \}$ if only if $\models_{N} \{ P \} \; \text{body}(T@m) \; \{ Q_n, Q_e \}$

- If $s$ is the virtual routine $T:m$, then for all receivers $x$ such that $\tau(x) = T$ $\models_{N} \{ P \} \; T:m \; \{ Q_n, Q_e \}$ if only if $\models_{N} \{ P \} \; \text{imp}(\tau(x), m) \; \{ Q_n, Q_e \}$

The above definition presents the semantics for Hoare Triples with empty assumptions. The following definition introduces the semantics of sequent:

**Definition 6 (Sequent Holds).**

\[
\{ P^1 \} s_1 \{ Q^1_n, Q^1_e \}, ..., \{ P^j \} s_j \{ Q^j_n, Q^j_e \} \models \{ P \} \; s \; \{ Q_n, Q_e \} \text{ if only if:}
\]

- for all $N$: $\models_{N} \{ P^1 \} s_1 \{ Q^1_n, Q^1_e \}$ and ... and $\models_{N} \{ P^j \} s_j \{ Q^j_n, Q^j_e \}$ implies

The above definition presents the semantics for Hoare Triples with empty assumptions. The following definition introduces the semantics of sequent:

**Theorem 9 (Soundness Theorem).**

\[ A \vdash \{ P \} \; s \; \{ Q_n, Q_e \} \Rightarrow A \models \{ P \} \; s \; \{ Q_n, Q_e \} \]

**Theorem 10 (Completeness Theorem).**

\[ \models \{ P \} \; s \; \{ Q_n, Q_e \} \Rightarrow A \vdash \{ P \} \; s \; \{ Q_n, Q_e \} \]

The following sections present the proofs of the soundness and completeness theorems.
B.2 Soundness Proof

The followings auxiliary lemmas are used to prove soundness:

Lemma 5 (Triple \( \models \), Triple \( \models_{N} \)).
\[
\models \{ P \} s \{ Q_{n} , Q_{e} \} \text{ if only if } \forall N : \models_{N} \{ P \} s \{ Q_{n} , Q_{e} \}
\]

Proof. By induction on the length of the execution trace.

Lemma 6 (Monotone \( \rightarrow_{N} \)).
\[
\langle \sigma, s_{1} \rangle \rightarrow_{N} \sigma', \chi \Rightarrow \langle \sigma, s_{1} \rangle \rightarrow_{N+1} \sigma', \chi
\]

Lemma 7 (\( \rightarrow \) iff \( \rightarrow_{N} \)).
\[
\langle \sigma, s_{1} \rangle \rightarrow \sigma', \chi \text{ if only if } \exists N : \langle \sigma, s_{1} \rangle \rightarrow_{N} \sigma', \chi
\]

Lemma 8 (Monotone \( \models_{N} \)).
\[
\models_{N+1} \{ P \} s \{ Q_{n} , Q_{e} \} \text{ implies } \models_{N} \{ P \} s \{ Q_{n} , Q_{e} \}
\]

The proof of soundness runs by induction on the structure of the derivation tree for:
\[
A \models \{ P \} s \{ Q_{n} , Q_{e} \}
\]

and the operational semantics.

Following, we present the soundness proof for the most interesting rules.

B.2.1 Assignment Axiom

We have to prove:
\[
\vdash \left\{ \begin{array}{l}
(safe(e) \land P[e/x]) \lor (\neg safe(e) \land Q_{e}) \\
(\neg safe(e) \land Q_{e})
\end{array} \right\} x := e \{ P , Q_{e} \} \text{ implies }
\]
\[
\models \left\{ \begin{array}{l}
(safe(e) \land P[e/x]) \lor (\neg safe(e) \land Q_{e}) \\
(\neg safe(e) \land Q_{e})
\end{array} \right\} x := e \{ P , Q_{e} \}
\]

Let \( P' \) be defined as \( P' \triangleq (safe(e) \land P[e/x]) \lor (\neg safe(e) \land Q_{e}) \).

Applying Definition 5 and Definition 6 to the consequence of the rule, we have to prove:
∀σ ⊨ P' : (σ, x := e) →_N σ', χ then

χ = normal ⇒ σ' ⊨ P, and
χ = exc ⇒ σ' ⊨ Q_e

We prove it doing case analysis on χ:

Case 1: χ = exc. By the definition of the operational semantics, we have:

\[
\sigma(e) = \text{exc}
\]

Thus, we have to prove \(\sigma \mid= Q_e\). Since \(\sigma(e) = \text{exc}\), applying Lemma 1, we know \(\sigma \mid= \neg\text{safe}(e)\). Since \(\sigma \mid= P'\), and \(\sigma\) does not change, then \(\sigma \mid= Q_e\).

Case 2: χ = normal. By the definition of the operational semantics, we get:

\[
\sigma(e) \neq \text{exc}
\]

Thus, we have to prove \(\sigma[x := \sigma(e)] \mid= P\). Since \(\sigma(e) \neq \text{exc}\), applying Lemma 1, we know \(\sigma \mid= \text{safe}(e)\). Since \(\sigma \mid= P'\), then \(\sigma \mid= \text{safe}(e) \land P[e/x]\). Applying Lemma 2, then \(\sigma[x := \sigma(e)] \mid= P\).

\(\Box\)

B.2.2 Compound Rule

We have to prove:

\[\mathcal{A} \vdash \{ P \} \; s_1; s_2 \{ R_n, R_e \} \implies \mathcal{A} \vdash \{ P \} \; s_1; s_2 \{ R_n, R_e \}\]

using the induction hypotheses:

\[\mathcal{A} \vdash \{ P \} \; s_1 \{ Q_n, R_e \} \implies \mathcal{A} \vdash \{ P \} \; s_1 \{ Q_n, R_e \}\]

\[\mathcal{A} \vdash \{ Q_n \} \; s_2 \{ R_n, R_e \} \implies \mathcal{A} \vdash \{ Q_n \} \; s_2 \{ R_n, R_e \}\]

Let \(\mathcal{A}\) be \(\mathcal{A}_1, ..., \mathcal{A}_n\) where \(\mathcal{A}_1, ..., \mathcal{A}_N\) are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

\[\text{for all } N : \models_N \mathcal{A}_1, \text{and} ..., \models_N \mathcal{A}_n \implies \models_N \{ P \} \; s_1; s_2 \{ R_n, R_e \}\]

using the hypotheses:
for all $N : \models_N A_1, \ldots, \models_N A_n$ implies $\models_N \{ P \} s_1 \{ Q_n, R_e \}$

for all $N : \models_N A_1, \ldots, \models_N A_n$ implies $\models_N \{ Q_n \} s_2 \{ R_n, R_e \}$

Since the sequent $A$ is the same in the hypotheses and the conclusion, and since $s_1$ and $s_2$ are instructions, applying Definition 5, we have to show:

for all $\sigma : \models_P : \langle \sigma, s_1; s_2 \rangle \rightarrow_N \sigma'', \chi$ then

$\chi = normal \Rightarrow \sigma'' \models R_n, \text{ and}$

$\chi = exc \Rightarrow \sigma'' \models R_e$ (B.2)

using the hypotheses:

for all $\sigma : \models_P : \langle \sigma, s_1 \rangle \rightarrow_N \sigma', \chi$ then

$\chi = normal \Rightarrow \sigma' \models Q_n, \text{ and}$

$\chi = exc \Rightarrow \sigma' \models R_e$ (B.3)

and

for all $\sigma' : \models_Q : \langle \sigma', s_2 \rangle \rightarrow_N \sigma'', \chi$ then

$\chi = normal \Rightarrow \sigma'' \models R_n, \text{ and}$

$\chi = exc \Rightarrow \sigma'' \models R_e$ (B.4)

We prove it doing case analysis on $\chi$:

Case 1: $\chi = exc$. By the definition of the operational semantics for compound we get:

$\langle \sigma, s_1 \rangle \rightarrow_N \sigma', exc$

$\langle \sigma, s_1 ; s_2 \rangle \rightarrow_N \sigma', exc$

Since $\sigma \models P$, then by the first hypothesis [B.3] we get $\sigma' \models R_e$.

Case 2: $\chi = normal$. By the definition of the operational semantics for compound we have:

$\langle \sigma, s_1 \rangle \rightarrow_N \sigma', normal$

$\langle \sigma', s_2 \rangle \rightarrow_N \sigma'', \chi$

We can apply the first induction hypothesis (B.3) we get $\sigma' \models Q_n$ since $\sigma \models P$. Then, we can apply the second induction hypothesis (B.4) and get:

$\chi = normal \Rightarrow \sigma'' \models R_n, \text{ and } \chi = exc \Rightarrow \sigma'' \models R_e$

$\square$
B.2 Soundness Proof

B.2.3 Conditional Rule

We have to prove:

\[ A \vdash \{ P \} \text{ if } e \text{ then } s_1 \text{ else } s_2 \text{ end } \{ Q_n, Q_e \} \]

implies

\[ A \models \{ P \} \text{ if } e \text{ then } s_1 \text{ else } s_2 \text{ end } \{ Q_n, Q_e \} \]

using the induction hypotheses:

\[ A \vdash \{ P \wedge e \} \text{ } s_1 \{ Q_n, Q_e \} \]

implies

\[ A \models \{ P \wedge e \} \text{ } s_1 \{ Q_n, Q_e \} \]

\[ A \vdash \{ P \wedge \neg e \} \text{ } s_2 \{ Q_n, Q_e \} \]

implies

\[ A \models \{ P \wedge \neg e \} \text{ } s_2 \{ Q_n, Q_e \} \]

Let \( A \) be \( A_1, ..., A_n \) where \( A_1, ..., A_N \) are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

\[ \text{for all } N : \models_N A_1, \text{ and } ..., \models_N A_n \]

implies

\[ \models_N \{ P \} \text{ if } e \text{ then } s_1 \text{ else } s_2 \text{ end } \{ Q_n, Q_e \} \]

using the hypotheses:

\[ \text{for all } N : \models_N A_1, \text{ and } ..., \models_N A_n \text{ implies } \models_N \{ P \} \text{ } s_1 \{ Q_n, Q_e \} \]

\[ \text{for all } N : \models_N A_1, \text{ and } ..., \models_N A_n \text{ implies } \models_N \{ P \} \text{ } s_2 \{ Q_n, Q_e \} \]

Since the sequent \( A \) is the same in the hypotheses and the conclusions, and \( s_1 \) and \( s_2 \) are instructions, applying Definition 6 we have to prove:

\[ \forall \sigma \models P : \langle \sigma, \text{ if } e \text{ then } s_1 \text{ else } s_2 \text{ end} \rangle \rightarrow_N \sigma', \chi \text{ then } \]

\[ \chi = \text{ normal } \Rightarrow \sigma' \models Q_n, \text{ and } \]

\[ \chi = \text{ exc } \Rightarrow \sigma' \models Q_e \]

(B.5)

using the hypotheses:

\[ \forall \sigma \models (P \wedge e) : \langle \sigma, s_1 \rangle \rightarrow_N \sigma', \chi \text{ then } \]

\[ \chi = \text{ normal } \Rightarrow \sigma' \models Q_n, \text{ and } \]

\[ \chi = \text{ exc } \Rightarrow \sigma' \models Q_e \]

(B.6)

and

\[ \forall \sigma \models (P \wedge \neg e) : \langle \sigma, s_2 \rangle \rightarrow_N \sigma', \chi \text{ then } \]

\[ \chi = \text{ normal } \Rightarrow \sigma' \models Q_n, \text{ and } \]

\[ \chi = \text{ exc } \Rightarrow \sigma' \models Q_e \]

(B.7)
We prove this rule doing case analysis on \( \sigma(e) \):

**Case 1:** \( \sigma(e) = \text{true} \). If \( \sigma(e) = true \) then by the definition of the operational semantics we get:

\[
\begin{align*}
\langle \sigma, s_1 \rangle \to_N \sigma', \chi & \quad \sigma(e) = \text{true} \\
\langle \sigma, \text{if } e \text{ then } s_1 \text{ else } s_2 \text{ end} \rangle & \to_N \sigma', \chi
\end{align*}
\]

Then applying the first hypothesis (B.6) we prove \( \chi = \text{normal} \Rightarrow \sigma' \models Q_n \), and \( \chi = \text{exc} \Rightarrow \sigma' \models Q_e \).

**Case 2:** \( \sigma(e) = \text{false} \). If \( \sigma(e) = false \) then by the definition of the operational semantics we get:

\[
\begin{align*}
\langle \sigma, s_2 \rangle \to_N \sigma', \chi & \quad \sigma(e) = \text{false} \\
\langle \sigma, \text{if } e \text{ then } s_1 \text{ else } s_2 \text{ end} \rangle & \to_N \sigma', \chi
\end{align*}
\]

Then applying the second hypothesis (B.7) we prove \( \chi = \text{normal} \Rightarrow \sigma' \models Q_n \), and \( \chi = \text{exc} \Rightarrow \sigma' \models Q_e \).

\( \Box \)

### B.2.4 Check Axiom

We have to prove:

\[
\vdash \{ P \} \text{ check } e \text{ end } \{ (P \land e), (P \land \neg e) \} \quad \text{implies}
\]

\[\models \{ P \} \text{ check } e \text{ end } \{ (P \land e), (P \land \neg e) \} \]

Applying Definition [5] and Definition [6] to the consequence of the rule, we have to prove:

\[
\forall \sigma \models P : \langle \sigma, \text{ check } e \text{ end} \rangle \to_N \sigma, \chi \text{ then}
\chi = \text{normal} \Rightarrow \sigma' \models (P \land e), \text{ and} \n\chi = \text{exc} \Rightarrow \sigma' \models (P \land \neg e)
\]

(B.8)

To prove it, we do case analysis on \( \sigma(e) \):

**Case 1:** \( \sigma(e) = \text{true} \). By the definition of the operational semantics we have:

\[
\sigma(e) = \text{true}
\]

\[
\langle \sigma, \text{check } e \text{ end} \rangle \to_N \sigma, \text{normal}
\]

Since the state \( \sigma \) is unchanged then \( \sigma \models P \). Furthermore, \( \sigma(e) = true \) by this case analysis, then applying the definition of \( \models \) we prove \( \sigma \models (P \land e) \)
Case 2: $\sigma(e) = \text{false}$. By the definition of the operational semantics we have:

$$\sigma(e) = \text{false} \quad \over \langle \sigma, \text{check } e \text{ end} \rangle \rightarrow_N \sigma, \text{exc}$$

Similar to the above case, $\sigma \models P$ since the state is unchanged and $\sigma(e) = \text{false}$ by the case analysis. Then $\sigma \models (P \land \neg e)$ holds using the definition of $\models$.

\[ \Box \]

### B.2.5 Loop Rule

We have to prove:

$$A \models \{ I \} \quad \text{until } e \text{ loop } s_1 \text{ end } \{ (I \land e), R_e \} \text{ implies } A \models \{ I \} \quad \text{until } e \text{ loop } s_1 \text{ end } \{ (I \land e), R_e \}$$

using the induction hypotheses:

$$A \models \{ \neg e \land I \} \quad s_1 \quad \{ I, R_e \} \text{ implies } A \models \{ \neg e \land I \} \quad s_1 \quad \{ I, R_e \} \quad I \Rightarrow I'$$

Let $A$ be $A_1, ..., A_n$ where $A_1, ..., A_N$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

for all $N : \models_N A_1, \ldots, A_n$ implies $\models_N \{ P \} \quad \text{from } \ldots \{ (I \land e), R_e \}$

using the hypothesis:

for all $N : \models_N A_1, \ldots, A_n$ implies $\models_N \{ \neg e \land I \} \quad s_1 \quad \{ I, R_e \}$

Since the sequent $A$ is the same in the hypothesis and the conclusions, and $s_1$ is an instruction, applying Definition 5 we have to prove:

$$\forall \sigma \models P : \forall \sigma \models P : \langle \sigma, \text{until } e \text{ loop } s_1 \text{ end} \rangle \rightarrow_N \sigma', \chi \text{ then}$$

$$\chi = \text{normal} \quad \Rightarrow \quad \sigma' \models (I \land e), \text{ and}$$
$$\chi = \text{exc} \quad \Rightarrow \quad \sigma' \models R_e$$

using the hypothesis:

$$\forall \sigma \models (\neg e \land I) : \langle \sigma, s_2 \rangle \rightarrow_N \sigma', \chi \text{ then}$$

$$\chi = \text{normal} \quad \Rightarrow \quad \sigma' \models I, \text{ and}$$
$$\chi = \text{exc} \quad \Rightarrow \quad \sigma' \models R_e \quad (B.9)$$
We prove this rule doing case analysis on $\sigma(e)$:

**Case 1: $\sigma(e) = \text{true}$**. By the operational semantics, we get:

$$
\begin{align*}
\sigma(e) &= \text{true} \\
\langle \sigma, \text{until e loop } s_1 \text{ end} \rangle &\rightarrow_N \sigma, \text{normal}
\end{align*}
$$

Applying the hypothesis we know $\sigma \models I$ and $\sigma \models e$. Then by the definition of $\models$ we prove $\sigma \models (I \land e)$.

**Case 2: $\sigma(e) = \text{false}$**. Then we do case analysis on $\chi$:

**Case 2.a: $\chi = \text{exc}$**. By the definition of the operational semantics we have:

$$
\begin{align*}
\sigma(e) &= \text{false} \\
\langle \sigma, s_1 \rangle &\rightarrow_N \sigma', \text{exc} \\
\langle \sigma', \text{until e loop } s_1 \text{ end} \rangle &\rightarrow_N \sigma', \text{exc}
\end{align*}
$$

By the first hypothesis we know $\sigma \models I$. Then since $\sigma(e) = \text{false}$ and $\chi = \text{exc}$, we prove $\sigma' \models R_e$.

**Case 2.b: $\sigma(e) = \text{false}$ and $\chi = \text{normal}$**. By the definition of the operational semantics we have:

$$
\begin{align*}
\sigma(e) &= \text{false} \\
\langle \sigma, s_1 \rangle &\rightarrow_N \sigma', \text{normal} \\
\langle \sigma', \text{until e loop } s_1 \text{ end} \rangle &\rightarrow_N \sigma'', \chi \\
\langle \sigma, \text{until e loop } s_1 \text{ end} \rangle &\rightarrow_N \sigma'', \chi
\end{align*}
$$

By the hypothesis we know $\sigma \models I$. Then since $\sigma(e) = \text{false}$, we can apply the definition of $\models$ and the second hypothesis (B.9), and we get $\sigma' \models I$. Now we can apply the induction hypothesis and prove

$$
\chi = \text{normal} \implies \sigma'' \models (I \land e), \text{ and } \chi = \text{exc} \implies \sigma'' \models R_e
$$

$\Box$

### B.2.6 Read Attribute Axiom

We have to prove:

$$
\begin{align*}
\vdash &\left\{ (y \neq \text{Void} \land P[\text{instvar}(y, T@a)/x]) \lor (y = \text{Void} \land Q_e) \right\} \quad x := y.T@a \quad \{ P, Q_e \} \quad \text{implies} \\
\vdash &\left\{ (y \neq \text{Void} \land P[\text{instvar}(y, T@a)/x]) \lor (y = \text{Void} \land Q_e) \right\} \quad x := y.T@a \quad \{ P, Q_e \}
\end{align*}
$$
Applying Definition 5 and Definition 6 to the consequence of the rule, we have to prove:

\[ \forall \sigma \models P' : \langle \sigma, x := y.T@a \rangle \rightarrow_N \sigma', \chi \text{ then} \]
\[ \chi = \text{normal} \Rightarrow \sigma' \models P, \text{ and} \]
\[ \chi = \text{exc} \Rightarrow \sigma' \models Q_e \]  
(B.10)

where \( P' \) is defined as follows:

\[ P' \equiv (y \neq \text{Void} \land P[S(instvar(y, T@a))/x]) \lor (y = \text{Void} \land Q_e) \]

To prove it, we do case analysis on \( \chi \):

**Case 1: \( \chi = \text{normal} \).** Applying the definition of the operational semantics we have:

\[ \sigma(y) \neq \text{voidV} \]

Then applying Lemma 2 we get \( \sigma \models P \).

**Case 2: \( \chi = \text{exc} \).** Applying the definition of the operational semantics:

\[ \sigma(y) = \text{voidV} \]

we get \( \sigma \models Q_e \).

\[ \Box \]

### B.2.7 Write Attribute Axiom

We have to prove:

\[ \models \left\{ \begin{array}{l} (y \neq \text{Void} \land P[S(instvar(y, T@a) := e)/y]) \lor \\ (y = \text{Void} \land Q_e) \end{array} \right\} \quad \text{implies} \quad \left\{ \begin{array}{l} y.T@a := e \\ \{ P, Q_e \} \end{array} \right\} \]

\[ \models \left\{ \begin{array}{l} (y \neq \text{Void} \land P[S(instvar(y, T@a) := e)/y]) \lor \\ (y = \text{Void} \land Q_e) \end{array} \right\} \quad \text{implies} \quad \left\{ \begin{array}{l} y.T@a := e \\ \{ P, Q_e \} \end{array} \right\} \]

Applying Definition 5 and Definition 6 to the consequence of the rule, we have to prove:

\[ \forall \sigma \models P' : \langle \sigma, y.T@a := e \rangle \rightarrow_N \sigma', \chi \text{ then} \]
\[ \chi = \text{normal} \Rightarrow \sigma' \models P, \text{ and} \]
\[ \chi = \text{exc} \Rightarrow \sigma' \models Q_e \]  
(B.11)
where \( P' \) is defined as follows:

\[
P' \equiv (y \neq \text{Void} \land P \langle \text{instvar}(y, T\@a) := e \rangle / \$ ) \lor (y = \text{Void} \land Q_e)
\]

To prove it, we do case analysis on \( \chi \):

**Case 1:** \( \chi = \text{normal} \). Applying the definition of the operational semantics we have:

\[
\frac{\sigma(y) \neq \text{voidV}}{\langle \sigma, y.T\@a := e \rangle \rightarrow_N \sigma[\$ := \sigma(\$)(\text{instvar}(\sigma(y), T\@a) := \sigma(e))]}, \text{normal}
\]

Then applying Lemma 2 we get \( \sigma \models P \).

**Case 2:** \( \chi = \text{exc} \). Applying the definition of the operational semantics:

\[
\frac{\sigma(y) = \text{voidV}}{\langle \sigma, y.T\@a := e \rangle \rightarrow_N \sigma, \text{exc}}
\]

we get \( \sigma \models Q_e \).

\( \square \)

### B.2.8 Local Rule

We have to prove:

\[
\begin{align*}
\mathcal{A} \vdash \{ P \} & \quad \text{local } v_1 : T_1; \ldots v_n : T_n; s \{ Q_n, Q_e \} & \text{implies } \\
\mathcal{A} \models \{ P \} & \quad \text{local } v_1 : T_1; \ldots v_n : T_n; s \{ Q_n, Q_e \}
\end{align*}
\]

using the induction hypothesis:

\[
\begin{align*}
\mathcal{A} \vdash \{ P, v_1 = \text{init}(T_1) \land \ldots \land v_n = \text{init}(T_n) \} & \quad s \{ Q_n, Q_e \} & \text{implies } \\
\mathcal{A} \models \{ P, v_1 = \text{init}(T_1) \land \ldots \land v_n = \text{init}(T_n) \} & \quad s \{ Q_n, Q_e \}
\end{align*}
\]

Let \( \mathcal{A} \) be the sequent \( \mathcal{A} = \mathcal{A}_1, \ldots, \mathcal{A}_n \) where \( \mathcal{A}_1, \ldots, \mathcal{A}_n \) are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

for all \( N : \models_N \mathcal{A}_1, \ldots, \models_N \mathcal{A}_n \) implies \( \models_N \{ P \} \quad \text{local } v_1 : T_1; \ldots v_n : T_n; s \{ Q_n, Q_e \} \)

using the hypothesis:

\[
\begin{align*}
\models_N \{ P, v_1 = \text{init}(T_1) \land \ldots \lor v_n = \text{init}(T_n) \} & \quad s \{ Q_n, Q_e \}
\end{align*}
\]

Since the sequent \( \mathcal{A} \) is the same in the hypothesis and the conclusion, and since \( s \) is an instruction, applying Definition 5 we have to show:
∀σ ⊨ P : ⟨σ, local v_1 : T_1; ... v_n : T_n; s⟩ →_NS′, χ then

χ = normal ⇒ σ′ |= Q_n, and
χ = exc ⇒ σ′ |= Q_e

using the hypothesis:

∀σ ⊨ P′ : ⟨σ, s⟩ →_N σ′, χ then
χ = normal ⇒ σ′ |= Q_n, and
χ = exc ⇒ σ′ |= Q_e

(B.12)

where P′ is defined as follows:

P′ ≜ P ∧ v_1 = init(T_1) ∧ ... ∧ v_n = init(T_n)

Applying the definition of the operational semantics for locals, we get:

⟨σ[v_1 := init(T_1), ..., v_n := init(T_n)], s⟩ →_N σ′, normal

Then, applying Lemma 2 we know σ |= P′. Finally, applying the hypothesis (B.12) we prove:

χ = normal ⇒ σ′ |= Q_n, and χ = exc ⇒ σ′ |= Q_e

□

B.2.9 Creation Rule

We have to prove:

\[ \mathcal{A} ⊢ \begin{cases} P & \text{new}(\$, T)/Current, \\ T : make & Q_n, Q_e \end{cases} \implies \mathcal{A} ⊨ \begin{cases} P & \text{create} \{ T \}.make(e) & Q_n[x/Current] \implies Q_n, Q_e \end{cases} \]

using the induction hypothesis:

\[ \mathcal{A} \vdash \begin{cases} P & T : make & Q_n, Q_e \end{cases} \implies \mathcal{A} \models \begin{cases} P & T : make & Q_n, Q_e \end{cases} \]

Applying Definition 5 and Definition 6 to the consequence of the rule, we have to prove:
∀σ ⊨ P′ : ⟨σ, x := create {T}.make(e)⟩ →_{N+1} σ′, χ then

χ = normal ⇒ σ′ ⊨ Q_n[x/Current], and
χ = exc ⇒ σ′ ⊨ Q_e

where P′ is defined as follows:

P′ ≜ P [ new($, T)/Current, $⟨T⟩/$, e/p ]

using the hypothesis:

∀σ ⊨ P : ⟨σ, body(imp(T, make))⟩ →_{N} σ′, χ then

χ = normal ⇒ σ′ ⊨ Q_n, and
χ = exc ⇒ σ′ ⊨ Q_e

We prove soundness of this rule with respect to the operational semantics of creation instruction (defined in Section 3.1.3 on page 26) for an arbitrary N.

Since σ ⊨ P[new($, T)/Current, $⟨T⟩/$, e/p], then by Lemma 2 we know

σ[Current := new(σ($), T), $ := σ($)(T), p := σ(e)] ⊨ P

Applying the definition of the operational semantics we get

χ = normal ⇒ σ′[x := σ′(Current)] ⊨ Q_n, and
χ = exc ⇒ σ′ ⊨ Q_e

Using Lemma 2 we prove:

χ = normal ⇒ σ′ ⊨ Q_n[x/Current], and χ = exc ⇒ σ′ ⊨ Q_e

□

B.2.10 Rescue Rule

We have to prove:

\[ \mathcal{A} \vdash \{ P \} \quad s_1 \text{ rescue } s_2 \quad \{ Q_n, R_e \} \quad \text{implies} \quad \mathcal{A} \models \{ P \} \quad s_1 \text{ rescue } s_2 \quad \{ Q_n, R_e \} \]

using the induction hypotheses:

\[ \mathcal{A} \vdash \{ I_r \} \quad s_1 \quad \{ Q_n, Q_e \} \quad \text{implies} \quad \mathcal{A} \vdash \{ I_r \} \quad s_1 \quad \{ Q_n, Q_e \} \quad \text{and} \]
\[ \mathcal{A} \vdash \{ Q_e \} \quad s_2 \quad \{ (\text{Retry} \Rightarrow I_r) \land (\neg \text{Retry} \Rightarrow R_e), R_e \} \quad \text{implies} \quad \mathcal{A} \models \{ Q_e \} \quad s_2 \quad \{ (\text{Retry} \Rightarrow I_r) \land (\neg \text{Retry} \Rightarrow R_e), R_e \} \quad \text{and} \]
\[ P \Rightarrow I_r \]
Let $A$ be $A_1, \ldots, A_n$ where $A_1, \ldots, A_N$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

$$\text{for all } N : \models_N A_1, \text{ and } \ldots, \models_N A_n \text{ implies } \models_N \{ P \} \ s_1 \text{ rescue } s_2 \ \{ Q_n, R_e \}$$

using the hypotheses:

$$\text{for all } N : \models_N A_1, \text{ and } \ldots, \models_N A_n \text{ implies } \models_N \{ I_r \} \ s_1 \ \{ Q_n, Q_e \}$$

$$\text{for all } N : \models_N A_1, \text{ and } \ldots, \models_N A_n \text{ implies } \models_N \{ Q_e \} \ s_2 \ \{ (\text{Retry } \Rightarrow I_r) \land \neg (\text{Retry } \Rightarrow R_e), R_e \}$$

Since the sequent $A$ is the same in the hypotheses and the conclusions, and $s_1$ and $s_2$ are instructions, applying Definition 5 we have to prove:

$$\forall \sigma \models P : (\sigma, s_1 \text{ rescue } s_2) \rightarrow_N \sigma', \chi$$

using the hypotheses:

$$\forall \sigma \models I_r : (\sigma, s_1) \rightarrow_N \sigma', \chi$$

$$\chi = \text{normal } \Rightarrow \sigma' \models Q_n, \text{ and }$$

$$\chi = \text{exc } \Rightarrow \sigma' \models Q_e$$

using the hypotheses:

$$\forall \sigma \models Q_e : (\sigma, s_2) \rightarrow_N \sigma'', \chi$$

$$\chi = \text{normal } \Rightarrow \sigma'' \models ((\text{Retry } \Rightarrow I_r) \land \neg (\text{Retry } \Rightarrow R_e)), \text{ and }$$

$$\chi = \text{exc } \Rightarrow \sigma'' \models R_e$$

We prove this rule doing case analysis on $\chi$:

**Case 1: $\chi = \text{normal}$**: Since $s_1$ terminates normally, by the definition of the operational semantics we get:

$$\langle \sigma, s_1 \rangle \rightarrow_N \sigma', \text{ normal}$$

Then we can apply the first hypothesis (B.13) since $P \Rightarrow I_r$. Thus, we prove $\sigma' \models Q_n$. 

(B.13)
Case 2: $\chi = \text{exc}$. If $s_1$ triggers an exception, then by the operational semantics we have:

$$\langle \sigma, s_1 \rangle \rightarrow_N \sigma', \text{exc}$$

We have several cases depending if $\text{Retry}$ evaluates to true or not, and if $s_2$ terminates normally or not:

Case 2.a: $\chi = \text{exc}$. By the definition of the operational semantics we have:

$$\langle \sigma, s_1 \rangle \rightarrow_N \sigma', \text{exc} \quad \langle \sigma', s_2 \rangle \rightarrow_N \sigma'', \text{exc}$$

By the first hypothesis (B.13), we prove $\sigma' \models Q_e$. Then, we can apply the second hypothesis (B.14), and we get $\sigma'' \models R_e$.

Case 2.b: $\chi = \text{normal}$. Here, we do case analysis on $\sigma''(\text{Retry})$:

Case 2.b.1: $\sigma''(\text{Retry}) = \text{false}$. By the definition of the operational semantics we have:

$$\langle \sigma, s_1 \rangle \rightarrow_N \sigma', \text{exc} \quad \langle \sigma', s_2 \rangle \rightarrow_N \sigma'', \text{normal} \quad \neg \sigma''(\text{Retry})$$

By the first hypothesis (B.13), we prove $\sigma' \models Q_e$. Then, we can apply the second hypothesis (B.14) and we get $\sigma'' \models (\text{Retry} \Rightarrow I_r) \land (\neg \text{Retry} \Rightarrow R_e)$. Since $\sigma''(\text{Retry}) = \text{false}$ then by the definition of $\models$ we prove $\sigma'' \models R_e$.

Case 2.b.2: $\sigma''(\text{Retry}) = \text{true}$. The definition of the operational semantics is:

$$\langle \sigma, s_1 \rangle \rightarrow_N \sigma', \text{exc} \quad \langle \sigma', s_2 \rangle \rightarrow_N \sigma'', \text{normal} \quad \sigma''(\text{Retry}) \quad \langle \sigma'', s_1 \text{ rescue } s_2 \rangle \rightarrow_N \sigma''', \chi$$

Applying the first hypothesis (B.13), we prove $\sigma' \models Q_e$. Then, we can apply the second hypothesis (B.14), and we get $\sigma'' \models (\text{Retry} \Rightarrow I_r) \land (\neg \text{Retry} \Rightarrow R_e)$. Since $\sigma''(\text{Retry}) = \text{true}$ then by the definition of $\models$ we prove $\sigma'' \models I_r$. Now we can apply the induction hypothesis and we prove

$$\chi = \text{normal} \Rightarrow \sigma' \models Q_e, \text{ and } \chi = \text{exc} \Rightarrow \sigma' \models R_e$$

$\square$
B.2 Soundness Proof

B.2.11 Once Functions Rule

The proof of the rule for once functions (defined in Section 3.2.3 on page 45) is done in a similar way than the creation procedure. We use the once function rule and the invocation rule, and we prove they are sound with respect to the operation semantics of once (defined in Section 3.2.2 on page 38).

Let $P$ be the following precondition, where $T \_M \_RES$ is a logical variable:

$$P \equiv \begin{cases} \neg T_@m\_done \land P' \lor \\ T_@m\_done \land P'' \land T_@m\_result = T \_M \_RES \land \neg T_@m\_exc \lor \\ T_@m\_done \land P'' \land T_@m\_exc \end{cases}$$

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

$$Q'_n \equiv \begin{cases} T_@m\_done \land \neg T_@m\_exc \land \\ (Q_n \lor (P'' \land \text{Result} = T \_M \_RES \land T_@m\_result = T \_M \_RES)) \end{cases}$$

$$Q'_e \equiv \begin{cases} T_@m\_done \land T_@m\_exc \land (Q_e \lor P'') \end{cases}$$

To prove the once function rule, we have to prove:

$$A \vdash \{P\} \ T_@m \{Q'_n, Q'_e\} \implies$$

using the induction hypothesis:

$$A, \{P\} \ T_@m \{Q'_n, Q'_e\} \vdash$$

$$\begin{cases} P'[false/T_@m\_done] \land T_@m\_done \end{cases} \ \text{body}(T_@m) \ \{ (Q_n \land T_@m\_done), (Q_e \land T_@m\_done) \}$$

implies

$$A, \{P\} \ T_@m \{Q'_n, Q'_e\} \models$$

$$\begin{cases} P'[false/T_@m\_done] \land T_@m\_done \end{cases} \ \text{body}(T_@m) \ \{ (Q_n \land T_@m\_done), (Q_e \land T_@m\_done) \}$$

Let $A = A_1, ..., A_n$ where $A_1, ..., A_N$ are Hoare triples. By the semantics of sequents (Definition 3), we have to show:

$$\text{for all } N : \models_N A_1, \text{ and }, ..., \models_N A_n \implies \models_N \{P\} \ T_@m \{Q'_n, Q'_e\}$$
using the hypothesis:

\[
\text{for all } N : \models_N \mathcal{A}_1, \ldots, \models_N \mathcal{A}_n \text{ and } \models_N \{P\} \; T@m \{Q'_n, Q'_e\}
\]

implies

\[
\models_N \left\{ \begin{array}{l}
P'[\text{false}/T@m\text{.done}] \land \\
T@m\text{.done}
\end{array} \right\} \text{ \text{body}(T@m) \{ (Q_n \land T@m\text{.done}) , (Q_e \land T@m\text{.done}) \} }
\]

(B.15)

We prove it by induction on \(N\).

**Base Case:** \(N = 0\). Holds by Definition 5.

**Induction Case:** \(N \Rightarrow N + 1\). Assuming the induction hypothesis

\[
\models_N \mathcal{A}_1, \ldots, \models_N \mathcal{A}_n \text{ implies } \models_N \{P\} \; T@m \{Q'_n, Q'_e\}
\]

we have to show

\[
\models_{N+1} \mathcal{A}_1, \ldots, \models_{N+1} \mathcal{A}_n \text{ implies } \models_{N+1} \{P\} \; T@m \{Q'_n, Q'_e\}
\]

Then, we can prove this as follows:

\[
\models_{N+1} \mathcal{A}_1, \ldots, \models_{N+1} \mathcal{A}_n \text{ implies [Lemma 8]}
\]

\[
\models_N \mathcal{A}_1, \ldots, \models_N \mathcal{A}_n
\]

\[
\text{applying induction hypothesis}
\]

\[
\models_N \{ P \} \; T@m \{ Q'_n, Q'_e \}
\]

Using \(\models_N \mathcal{A}_1, \ldots, \models_N \mathcal{A}_n\), and \(\models_N \{ P \} \; T@m \{ Q'_n, Q'_e \}\), we can apply the hypothesis \(\text{[B.15]}\), and we get:

\[
\models_N \left\{ \begin{array}{l}
P'[\text{false}/T@m\text{.done}] \land \\
T@m\text{.done}
\end{array} \right\} \text{ \text{body}(T@m) \{ (Q_n \land T@m\text{.done}) , (Q_e \land T@m\text{.done}) \} }
\]

(B.16)

Since we know \(\text{[B.16]}\) holds, we can prove \(\models_{N+1} \{ P \} \; T@m \{ Q'_n, Q'_e \}\) using the hypothesis \(\text{[B.16]}\). Applying Definition 5, we have to prove

\[
\models_N \{ P \} \; \text{body}(T@m) \{ Q'_n, Q'_e \}
\]

assuming
B.2 Soundness Proof

\[ \models_N \left\{ \begin{array}{l} P'[\false / T@m_{\text{done}}] \\ T@m_{\text{done}} \end{array} \right\} \text{body}(T@m) \ \{ (Q_n \land T@m_{\text{done}}), (Q_e \land T@m_{\text{done}}) \} \]

Then, applying Definition 5 (\[\models_N\]), we have to prove:
\[ \forall \sigma \models P : (\sigma, \text{body}(T@m)) \rightarrow_N \sigma', \chi \text{ then} \]
\[ \chi = \text{normal} \Rightarrow \sigma' \models Q'_n, \text{ and} \]
\[ \chi = \text{exc} \Rightarrow \sigma' \models Q'_e \]

using the hypothesis:
\[ \forall \sigma \models P'[\false / T@m_{\text{done}}] \land T@m_{\text{done}} : (\sigma, \text{body}(T@m)) \rightarrow_N \sigma', \chi \text{ then} \]
\[ \chi = \text{normal} \Rightarrow \sigma' \models Q_n \land T@m_{\text{done}}, \text{ and} \]
\[ \chi = \text{exc} \Rightarrow \sigma' \models Q_e \land T@m_{\text{done}} \]  \(\text{(B.17)}\)

We prove this with respect to the operational semantics of once routines (defined in page 38) by case analysis on \(\chi\) and \(T@m_{\text{done}}\):

Case 1: \(\sigma(T@m_{\text{done}}) = \false\) and \(\chi = \text{normal}\). By the definition of the operational semantics we have:

\[ T'@m = \text{impl}(\tau(\sigma(y)), m) \quad T'@m \text{ is a once routine} \]
\[ \sigma(T'@m_{\text{done}}) = \false \]
\[ \langle \sigma[T'@m_{\text{done}} := \true, \text{Current} := y, p := \sigma(e)], \text{body}(T'@m) \rangle \rightarrow_N \sigma', \text{normal} \]
\[ (\sigma, x := y, S : m(e)) \rightarrow_{N+1} \sigma'[x := \sigma'(\text{Result})], \text{normal} \]

First, we show that \(T'@m = T@m\) because the operational semantics assigns to \(T'@m\) and the rule uses \(T@m\). Since the rule is derived applying the invocation rule, and the class rule, we know \(T@m = \text{impl}(T, m)\) and \(\tau(\text{Current}) = T\). However, \(T'@m = \text{impl}(\tau(y), m)\), and we now \(\tau(y) = \tau(\text{Current})\), then we can conclude that \(T@m = T'@m\).

Then, \(\sigma \models P'[\false / T@m_{\text{done}}] \land T@m_{\text{done}} \text{ because} \)
\[ \sigma[T'@m_{\text{done}} := \true, \text{Current} := y, p := \sigma(e)] \models P' \]

Now, we can apply the induction hypothesis \(\text{B.17}\) and we get \(\sigma' \models Q_n \land T@m_{\text{done}}\). Since \(Q_n \land T@m_{\text{done}} \Rightarrow Q'_n\) then \(\sigma' \models Q'_n\).

Case 2: \(\sigma(T@m_{\text{done}}) = \false\) and \(\chi = \text{exc}\). By the definition of the operational semantics we have:
\( T@m = \text{impl}(\tau(\sigma(y)), m) \quad T@m \text{ is a once routine} \)

\[
\begin{align*}
\sigma(T@m_{\text{done}}) &= \text{false} \\
\langle \sigma[T@m_{\text{done}} := \text{true}, \text{Current} := y, p := \sigma(e)], \text{body}(T@m) \rangle &\rightarrow_N \sigma', \text{exc} \\
\langle \sigma, x := y.S:m(e) \rangle &\rightarrow_{N+1} \sigma'[T@m_{\text{exc}} := \text{true}], \text{exc}
\end{align*}
\]

Applying a similar reasoning to Case 1, we know \( T'@m = T@m \). Since:

\[
\sigma \models P'[\text{false}/T@m_{\text{done}}] \land T@m_{\text{done}}
\]

because \( \sigma[T@m_{\text{done}} := \text{true}, \text{Current} := y, p := \sigma(e)] \models P' \), we can apply the induction hypothesis [B.17] and we get \( \sigma' \models Q_e \land T@m_{\text{done}} \). Then \( \sigma' \models Q'_e \) because \( Q_e \land T@m_{\text{done}} \Rightarrow Q'_e \).

**Case 3:** \( \sigma(T@m_{\text{done}}) = \text{true} \) and \( \chi = \text{normal} \). The definition of the operational semantics is the following:

\[
\begin{align*}
T@m &= \text{impl}(\tau(\sigma(y)), m) \quad T@m \text{ is a once routine} \\
\sigma(T@m_{\text{done}}) &= \text{true} \\
\sigma(T@m_{\text{exc}}) &= \text{false} \\
\langle \sigma, x := y.S:m(e) \rangle &\rightarrow_N \sigma[x := \sigma(T@m_{\text{result}})], \text{normal}
\end{align*}
\]

We know \( T'@m = T@m \). Since \( \sigma \models P \), and the state is unchanged except for the variable \( x \), and \( \sigma(T@m_{\text{done}}) = \text{true} \) and \( \sigma(T@m_{\text{exc}}) = \text{false} \), then \( \sigma \models P'' \). Then \( \sigma \models Q'_n \).

**Case 4:** \( \sigma(T@m_{\text{done}}) = \text{true} \) and \( \chi = \text{exc} \). By the definition of the operational semantics we have:

\[
\begin{align*}
T@m &= \text{impl}(\tau(\sigma(y)), m) \quad T@m \text{ is a once routine} \\
\sigma(T@m_{\text{done}}) &= \text{true} \\
\sigma(T@m_{\text{exc}}) &= \text{true} \\
\langle \sigma, x := y.S:m(e) \rangle &\rightarrow_N \sigma, \text{exc}
\end{align*}
\]

We know \( T'@m = T@m \). Since \( \sigma \models P \), and the state is unchanged, and \( \sigma(T@m_{\text{done}}) = \text{true} \) and \( \sigma(T@m_{\text{exc}}) = \text{true} \), then \( \sigma \models P'' \). Therefore, \( \sigma \models Q'_e \).

This concludes the proof.

\( \Box \)
B.2 Soundness Proof

B.2.12 Routine Implementation Rule

To prove this rule, we have to prove:

$$\vdash A \{ P \} \text{ body}(T@m) \{ Q_n, Q_e \}$$

using the induction hypothesis:

$$\vdash A \{ P \} \text{ body}(T@m) \{ Q_n, Q_e \}$$

Let $A$ be the sequent $A = A_1, ..., A_n$ where $A_1, ..., A_N$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

for all $N : \vdash_N A_1, and..., \vdash_N A_n$ implies $\vdash_N \{ P \} \text{ body}(T@m) \{ Q_n, Q_e \}$

We prove it by induction on $N$.

**Base Case**: $N = 0$. Holds by Definition 5.

**Induction Case**: $N \Rightarrow N + 1$. Assuming the induction hypothesis

$$\vdash_N A_1, and..., \vdash_N A_n$$

we have to show

$$\vdash_{N+1} A_1, and..., \vdash_{N+1} A_n$$

Then, we can prove this as follows:

$$\vdash_{N+1} A_1, and..., \vdash_{N+1} A_n$$

implies [Lemma 8]

applying induction hypothesis

Using $\vdash_N A_1, and..., \vdash_N A_n$, and $\vdash_N \{ P \} \text{ body}(T@m) \{ Q_n', Q_e' \}$, we can apply the hypothesis (B.18), and we get $\vdash_N \{ P \} \text{ body}(T@m) \{ Q_n, Q_e \}$. Then, by Definition 5 we prove $\vdash_{N+1} \{ P \} \text{ body}(T@m) \{ Q_n, Q_e \}$.

□
B.2.13 Routine Invocation Rule

To prove this rule, we have to prove:

\[ \mathcal{A} \vdash \{ (y \neq \text{Void} \land P[y/\text{Current}, e/p]) \lor (y = \text{Void} \land Q_e) \} \]
\[ x := y.T:m(e) \{ Q_n[x/\text{Result}] \land Q_e \} \]

implies

\[ \mathcal{A} \vdash \{ (y \neq \text{Void} \land P[y/\text{Current}, e/p]) \lor (y = \text{Void} \land Q_e) \} \]
\[ x := y.T:m(e) \{ Q_n[x/\text{Result}] \land Q_e \} \]

using the induction hypothesis:

\[ \mathcal{A} \vdash \{ P \} \quad T:m \quad \{ Q_n \land Q_e \} \quad \text{implies} \quad \mathcal{A} \vdash \{ P \} \quad T:m \quad \{ Q_n \land Q_e \} \]

Let \( \mathcal{A} \) be \( \mathcal{A}_1, \ldots, \mathcal{A}_n \) where \( \mathcal{A}_1, \ldots, \mathcal{A}_n \) are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

for all \( N : (\vdash_N \mathcal{A}_1, \text{and} \ldots, \vdash_N \mathcal{A}_n) \)
implies

\[ \vdash_N \{ (y \neq \text{Void} \land P[y/\text{Current}, e/p]) \lor (y = \text{Void} \land Q_e) \} \]
\[ x := y.T:m(e) \{ Q_n[x/\text{Result}] \land Q_e \} \]

using the hypothesis:

for all \( N : (\vdash_N \mathcal{A}_1, \text{and} \ldots, \vdash_N \mathcal{A}_n) \) implies \( \vdash_N \{ P \} \quad T:m \quad \{ Q_n \land Q_e \} \)

Let \( P' \) be \( (y \neq \text{Void} \land P[y/\text{Current}, e/p]) \lor (y = \text{Void} \land Q_e) \). Since the sequent \( \mathcal{A} \) is the same in the hypothesis and the conclusion, applying Definition 6 we have to show:

for all \( \sigma \models P' : (\sigma, x := y.T:m(e)) \rightarrow_N \sigma'' \land \chi \) then
\[ \chi = \text{normal} \Rightarrow \sigma'' \models Q_n[x/\text{Result}], \text{ and} \quad (B.19) \]
\[ \chi = \text{exc} \Rightarrow \sigma'' \models Q_e \]

using the hypothesis:

for all \( \sigma \models P : (\sigma, \text{body(imp}(\tau(\text{Current}), m)))) \rightarrow_{N-1} \sigma' \land \chi \) then
\[ \chi = \text{normal} \Rightarrow \sigma' \models Q_n, \text{ and} \quad (B.20) \]
\[ \chi = \text{exc} \Rightarrow \sigma' \models Q_e \]
We do case analysis on $\sigma(y)$:

**Case 1**: $\sigma(y) = \text{void}$. By the operational semantics we have:

$$T : m \text{ is not a once routine}$$

$$\sigma(y) = \text{void}$$

$$(\sigma, x := y. T : m(e)) \rightarrow_N \sigma, \text{exc}$$

Then, $\sigma \models Q_e$ since $\sigma \models P$ and $\chi = \text{exc}$.

**Case 2**: $\sigma(y) \neq \text{void}$. The definition of the operational semantics is:

$$T : m \text{ is not a once routine}$$

$$\sigma(y) \neq \text{void}$$

$$(\sigma, \text{Current} := \sigma(y), p := \sigma(e), \text{body}(\tau(\sigma(y)), m)) \rightarrow_{N+1} \sigma'(x := \sigma'(\text{Result})), \chi$$

Since $\sigma \models P'$, then applying Lemma 2, $\sigma \models P$. Then since $\text{Current} := \sigma(y)$, we can apply the induction hypothesis and Lemma 2 again, and we get

$$\chi = \text{normal} \Rightarrow \sigma'' \models Q_n[x/\text{Result}], \text{ and } \chi = \text{exc} \Rightarrow \sigma'' \models Q_e$$

\[\square\]

**B.2.14 Class Rule**

We have to prove:

$$A \vdash \{ \tau(\text{Current}) \preceq T \land P \} \quad T : m \quad \{ Q_n, Q_e \} \quad \text{implies}$$

$$A \models \{ \tau(\text{Current}) \preceq T \land P \} \quad T : m \quad \{ Q_n, Q_e \}$$

using the induction hypotheses:

$$A \vdash \{ \tau(\text{Current}) = T \land P \} \quad \text{imp}(T, m) \quad \{ Q_n, Q_e \} \quad \text{implies}$$

$$A \models \{ \tau(\text{Current}) = T \land P \} \quad \text{imp}(T, m) \quad \{ Q_n, Q_e \}$$

and

$$A \vdash \{ \tau(\text{Current}) < T \land P \} \quad T : m \quad \{ Q_n, Q_e \} \quad \text{implies}$$

$$A \models \{ \tau(\text{Current}) < T \land P \} \quad T : m \quad \{ Q_n, Q_e \}$$

Let $A$ be $A_1, \ldots, A_n$ where $A_1, \ldots, A_n$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

$$\text{for all } N : \models_N A_1, \ldots, \models_N A_n \text{ implies}$$

$$\models_N A \models \{ \tau(\text{Current}) \preceq T \land P \} \quad T : m \quad \{ Q_n, Q_e \}$$

using the hypotheses:
for all $N : \models_N A_1, \ldots, \models_N A_n$ implies
\[ \models_N A \models \{ \tau(Current) = T \land P \} \quad \text{imp}(T, m) \quad \{ Q_n, Q_e \} \]
and
\[ \models_N A \models \{ \tau(Current) \prec T \land P \} \quad T : m \quad \{ Q_n, Q_e \} \]

We prove it as follows:
\[ A \vdash \{ \tau(Current) \preceq T \land P \} \quad T : m \quad \{ Q_n, Q_e \} \]
\[ \Rightarrow \left[ \text{definition of } \tau \right] \]
\[ A \vdash \{ (\tau(Current) \prec T \lor \tau(Current) = T) \land P \} \quad T : m \quad \{ Q_n, Q_e \} \]
\[ \Rightarrow \left[ \text{hypothesis} \right] \]
\[ A \models \{ \tau(Current) = T \land P \} \quad \text{imp}(T, m) \quad \{ Q_n, Q_e \} \]
and
\[ A \models \{ \tau(Current) \prec T \land P \} \quad T : m \quad \{ Q_n, Q_e \} \]
\[ \Rightarrow \left[ \text{definition of } \models_N \right] \]
\[ A \models \{ \tau(Current) \preceq T \land P \} \quad T : m \quad \{ Q_n, Q_e \} \]
\[ \blacksquare \]

B.2.15 Subtype Rule

We have to prove:
\[ A \models \{ P \} \quad S : m \quad \{ Q_n, Q_e \} \quad \text{implies} \quad A \models \{ P \} \quad S : m \quad \{ Q_n, Q_e \} \]

using the induction hypotheses:
\[ \models_A \{ P \} \quad S : m \quad \{ Q_n, Q_e \} \quad \text{implies} \quad A \models \{ P \} \quad S : m \quad \{ Q_n, Q_e \} \]

We have to prove:
\[ A \models \{ \tau(Current) \preceq S \land P \} \quad T : m \quad \{ Q_n, Q_e \} \quad \text{iff} \left[ \text{definition of } \models_N \right] \]
\[ A \models \{ \tau(Current) \preceq S \land P \} \quad \text{imp}(\tau(Current), m) \quad \{ Q_n, Q_e \} \]
B.2 Soundness Proof

Since $\tau(\text{Current}) \preceq S$ and from the hypothesis we know $A \models \{ P \} \quad S.m \quad \{ Q_n, Q_e \}$, then applying the induction hypothesis we prove:

\[
A \models \{ \tau(\text{Current}) \preceq S \land P \} \quad T.m \quad \{ Q_n, Q_e \}
\]

\[\square\]

B.2.16 Language-Independent Rules

In this subsection, we prove the soundness of the language-independent rules presented in Section 3.1.4 on page 32.

Assumpt-axiom

We have to show that for all $N : \models_N A$ implies $\models_N A$, which is true.

\[\square\]

False-axiom

Let $A$ be $A_1, ..., A_n$ where $A_1, ..., A_n$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

for all $N : \models_N A_1, and..., \models_N A_n$ implies $\models_N \{ \text{false} \} \quad s \quad \{ \text{false}, \text{false} \}$

This holds by the definition of $\models_N$.

\[\square\]

Assumpt-intro-rule

Let $A$ be $A_1, ..., A_n$ where $A_1, ..., A_n$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

for all $N : \models_N A_1, and..., \models_N A_n, and A_0$ implies $\models_N A$

using the hypothesis:

for all $N : \models_N A_1, and..., \models_N A_n$ implies $\models_N A$

This holds by the hypothesis.

\[\square\]
Assumpt-elim-rule

Let $A$ be $A_1, ..., A_n$ where $A_1, ..., A_n$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

$$\text{for all } N : \models_N A_1, \text{ and }..., \models_N A_n \text{ implies } \models_N A$$

using the hypotheses:

$$\text{for all } N : \models_N A_1, \text{ and }..., \models_N A_n \text{ implies } \models_N A_0$$

$$\text{for all } N : \models_N A_1, \text{ and }..., \models_N A_n, \text{ and } A_0 \text{ implies } \models_N A$$

We prove it as follows:

$$\models_N A_1, \text{ and }..., \models_N A_n \quad (1)$$

$$\Rightarrow \underbrace{\left[ \text{applying the first hypothesis} \right]}_{\models_N A_0} \quad (2)$$

$$\Rightarrow \underbrace{\left[ \text{applying second hypothesis to (1) and (2)} \right]}_{\models_N A}$$

\[ \square \]

**Strength Rule**

We have to prove:

$$A \vdash \{ P' \} s_1 \{ Q_n, Q_e \} \text{ implies } A \models \{ P' \} s_1 \{ Q_n, Q_e \}$$

using the induction hypotheses:

$$A \vdash \{ P \} s_1 \{ Q_n, Q_e \} \text{ implies } A \models \{ P \} s_1 \{ Q_n, Q_e \}$$

and

$$P' \Rightarrow P$$

Applying the definition of $\models_N$, we have to prove:

$$\text{for all } \sigma \models P' : \langle \sigma, s_1 \rangle \rightarrow_N \sigma'', \chi \text{ then}$$

$$\chi = \text{normal} \Rightarrow \sigma'' \models Q_n, \text{ and}$$

$$\chi = \text{exc} \Rightarrow \sigma'' \models Q_e$$

Since $P' \Rightarrow P$, then we get $\sigma \models P$, then by hypothesis we prove:

$$A \models \{ P' \} s_1 \{ Q_n, Q_e \}$$

\[ \square \]
Weak Rule

We have to prove:

\[ A \vdash \{ P \} s_1 \{ Q'_n, Q'_e \} \text{ implies } A \models \{ P \} s_1 \{ Q'_n, Q'_e \} \]

using the induction hypotheses:

\[ A \vdash \{ P \} s_1 \{ Q_n, Q_e \} \text{ implies } A \models \{ P \} s_1 \{ Q_n, Q_e \} \]

and

\[ Q_n \Rightarrow Q'_n \]

\[ Q_e \Rightarrow Q'_e \]

Applying the definition of \( \models_N \), we have to prove:

for all \( \sigma \models P : (\sigma, s_1) \rightarrow_N \sigma'', \chi \) then

\[ \chi = \text{normal} \Rightarrow \sigma'' \models Q'_n, \text{ and} \]

\[ \chi = \text{exc} \Rightarrow \sigma'' \models Q'_e \]

Applying the hypothesis we get:

\[ \chi = \text{normal} \Rightarrow \sigma'' \models Q_n, \text{ and} \]

\[ \chi = \text{exc} \Rightarrow \sigma'' \models Q_e \]

Since \( Q_n \Rightarrow Q'_n \) and \( Q_e \Rightarrow Q'_e \) then we get

\[ \chi = \text{normal} \Rightarrow \sigma'' \models Q'_n, \text{ and} \]

\[ \chi = \text{exc} \Rightarrow \sigma'' \models Q'_e \]

and we prove \( A \models \{ P \} s_1 \{ Q'_n, Q'_e \} \)

\( \square \)

Conjunction Rule

We have to prove:

\[ A \vdash \{ P^1 \land P^2 \} s_1 \{ Q^1_n \land Q^2_n, Q^1_e \land Q^2_e \} \text{ implies } A \models \{ P^1 \land P^2 \} s_1 \{ Q^1_n \land Q^2_n, Q^1_e \land Q^2_e \} \]

using the induction hypotheses:

\[ A \vdash \{ P^1 \} s_1 \{ Q^1_n, Q^1_e \} \text{ implies } A \models \{ P^1 \} s_1 \{ Q^1_n, Q^1_e \} \]

\[ A \vdash \{ P^2 \} s_1 \{ Q^2_n, Q^2_e \} \text{ implies } A \models \{ P^2 \} s_1 \{ Q^2_n, Q^2_e \} \]

This holds applying the definition of \( \models_N \), and the hypotheses.

\( \square \)
Disjunction Rule

The proof is similar to the conjunction rule proof.

\[ \square \]

### B.3 Completeness Proof

As pointed out by Oheimb [124], the approach using weakest precondition cannot be used to prove completeness [30] of recursive routine calls. The postcondition of recursive routine calls changes such that the induction does not go through. Here, we use the Most General Formula (MGF) approach introduced by Gorelick [42], and promoted by Apt [3], Boer et al. [34], and others. The MGF of an instruction \( s \) gives the strongest postcondition for the most general precondition, which is the operational semantics of \( s \).

Following, we prove Theorem [10] by induction on the structure of the instruction \( s \). In this section, we present the proof for the most important cases.

**Lemma 9** (Completeness Routine Imp). Let \( \$ \) and \( \$' \) be object stores, and let \( \{ Q_n^{T@m}, Q_e^{T@m} \} \) be the strongest postcondition defined as follows:

\[
\{ Q_n^{T@m}, Q_e^{T@m} \} \triangleq SP(T@m, \$ = \$')
\]

Let \( A_0 \) be the sequent defined as follows:

\[
A_0 = \bigwedge_{T@m} \{ \$ = \$' \} \quad T@m \quad \{ Q_n^{T@m}, Q_e^{T@m} \}
\]

If \( \models \{ P \} \quad s \quad \{ Q_n, Q_e \} \) then \( A_0 \vdash \{ P \} \quad s \quad \{ Q_n, Q_e \} \)

Before proving Lemma 9, we use it to prove the completeness theorem.

**Lemma 10** (Sequent T@m).

\[
A, \{ \$ = \$' \} \quad T@m \quad \{ Q_n^{T@m}, Q_e^{T@m} \} \vdash \{ P \} \quad s \quad \{ Q_n, Q_e \} \quad \text{implies}
\]

\[
A \vdash \{ P \} \quad s \quad \{ Q_n, Q_e \}
\]

**Proof.** By induction on the derivation

\[
A \vdash \{ P \} \quad s \quad \{ Q_n, Q_e \}
\]

\[ \square \]
Now, we prove the completeness theorem:

**Proof of Completeness Theorem** We have to prove:

\[ \models \{ P \} \ s \ \{ Q_n, Q_e \} \Rightarrow \vdash \{ P \} \ s \ \{ Q_n, Q_e \} \]

Assume \( \models \{ P \} \ s \ \{ Q_n, Q_e \} \). Then, applying the Lemma 9 we get:

\[ \mathcal{A}_0 \vdash \{ P \} \ s \ \{ Q_n, Q_e \} \]

Then by repeated application of Lemma 10 we obtain:

\[ \vdash \{ P \} \ s \ \{ Q_n, Q_e \} \]

In the rest of this section, we prove Lemma 9 by induction on the measure of \( s \), defined as follows:

- If \( s \) is an instruction, the measure is the size of \( s \)
- The measure of \( T:m \) is 0
- The measure of \( T@m \) is \(-1\)

With this definition of measure, one can reason about instructions using induction hypotheses about their sub-parts, about a routine invocation using induction hypotheses of the form \( T:m \), and about \( T:m \) using induction hypotheses of the form \( T@m \).

**B.3.1 Assignment Axiom**

We have to prove:

\[ \models \left\{ (\text{safe}(e) \land P[e/x]) \lor (\neg\text{safe}(e) \land Q_e) \right\} \ x := e \ \{ P, Q_e \} \text{ implies} \]

\[ \mathcal{A}_0 \vdash \left\{ (\text{safe}(e) \land P[e/x]) \lor (\neg\text{safe}(e) \land Q_e) \right\} \ x := e \ \{ P, Q_e \} \]

Let \( P' \) be \( (\text{safe}(e) \land P[e/x]) \lor (\neg\text{safe}(e) \land Q_e) \).

Assume \( \models \{ P' \} \ x := e \ \{ Q'_n, Q'_e \} \), then \( \models P' \Rightarrow Q'_n[e/x] \). Then by the assignment axiom and the consequence rule we prove:

\[ \mathcal{A}_0 \vdash \{ P' \} \ x := e \ \{ Q'_n, Q'_e \} \]

\( \square \)
B.3.2 Compound Rule

We have to prove:

\[ \models \{ P \} \ s_1; s_2 \ \{ R_n, R_e \} \quad \text{implies} \quad \mathcal{A}_0 \vdash \{ P \} \ s_1; s_2 \ \{ R_n, R_e \} \]

using the hypotheses

\[ \models \{ P \} \ s_1 \ \{ Q_n, R_e \} \quad \text{implies} \quad \mathcal{A}_0 \vdash \{ P \} \ s_1 \ \{ Q_n, R_e \} \]

\[ \models \{ Q_n \} \ s_2 \ \{ R_n, R_e \} \quad \text{implies} \quad \mathcal{A}_0 \vdash \{ Q_n \} \ s_2 \ \{ R_n, R_e \} \]

Assume \( \models \{ P \} \ s_1; s_2 \ \{ R_n, R_e \} \). Then

\[ \models \{ P \} \ s_1 \ \{ T_n, T_e \} \quad \text{and} \quad \models \{ T_n \} \ s_2 \ \{ R'_n, R'_e \} \]

where \( \{ T_n, T_e \} \) and \( \{ R'_n, R'_e \} \) are the strongest postconditions defined as follows:

\[
\begin{align*}
\{ T_n, T_e \} &\triangleq s_1(P) \\
\{ R'_n, R'_e \} &\triangleq s_2(T_n)
\end{align*}
\]

By induction hypotheses we have:

\[ \mathcal{A}_0 \vdash \{ P \} \ s_1 \ \{ T_n, T_e \} \quad \text{and} \quad \mathcal{A}_0 \vdash \{ T_n \} \ s_2 \ \{ R'_n, R'_e \} \]

By the semantics of \( s_1; s_2 \) we have that \( R'_n \Rightarrow R_n \) and \( R'_e \Rightarrow R_e \) and \( T_e \Rightarrow R_e \). By the rule of consequence applied with implications \( R'_n \Rightarrow R_n \) and \( T_e \Rightarrow R_e \) and \( R'_e \Rightarrow R_e \), we obtain:

\[ \mathcal{A}_0 \vdash \{ P \} \ s_1 \ \{ T_n, R_e \} \quad \text{and} \quad \mathcal{A}_0 \vdash \{ T_n \} \ s_2 \ \{ R_n, R_e \} \]

The conclusion \( \mathcal{A}_0 \vdash \{ P \} \ s_1; s_2 \ \{ R_n, R_e \} \) follows by the compound rule.

\[\square\]

B.3.3 Conditional Rule

We have to prove:

\[ \models \{ P \} \ \text{if} \ e \ \text{then} \ s_1 \ \text{else} \ s_2 \ \text{end} \ \{ Q_n, Q_e \} \quad \text{implies} \quad \mathcal{A}_0 \vdash \{ P \} \ \text{if} \ e \ \text{then} \ s_1 \ \text{else} \ s_2 \ \text{end} \ \{ Q_n, Q_e \} \]

using the hypotheses

\[ \models \{ P \land e \} \quad s_1 \quad \{ Q_n \land Q_e \} \quad \text{implies} \quad \mathcal{A}_0 \vdash \{ P \land e \} \quad s_1 \quad \{ Q_n \land Q_e \} \]

and

\[ \models \{ Q_n \} \quad s_2 \quad \{ R_n \land R_e \} \quad \text{implies} \quad \mathcal{A}_0 \vdash \{ P \land \neg e \} \quad s_2 \quad \{ Q_n \land Q_e \} \]

Assume \( \models \{ P \} \) if \( e \) then \( s_1 \) else \( s_2 \) end \( \{ Q'_n \land Q'_e \} \). Then,

\[ \models \{ P \land e \} \quad s_1 \quad \{ Q'_n \land Q'_e \} \]

\[ \models \{ P \land \neg e \} \quad s_2 \quad \{ Q'_n \land Q'_e \} \]

where \( \{ T_n, T_e \} \) is the strongest postcondition \( \{ Q'_n, Q'_e \} \triangleq s_1(P \land e) \cup s_2(P \land \neg e) \)

Then by induction hypotheses, we know:

\[ \mathcal{A}_0 \vdash \{ P \land e \} \quad s_1 \quad \{ Q'_n \land Q'_e \} \]

\[ \mathcal{A}_0 \vdash \{ P \land \neg e \} \quad s_2 \quad \{ Q'_n \land Q'_e \} \]

Since \( Q'_n \Rightarrow Q_n \) and \( Q'_e \Rightarrow Q_e \), applying the conditional rule and the rule of consequence we obtain:

\[ \mathcal{A}_0 \vdash \{ P \} \quad \text{if} \quad e \quad \text{then} \quad s_1 \quad \text{else} \quad s_2 \quad \text{end} \quad \{ R_n \land R_e \} \]

\[ \square \]

B.3.4 Check Axiom

We have to prove:

\[ \models \{ P \} \quad \text{check} \quad e \quad \text{end} \quad \{ (P \land e), (P \land \neg e) \} \quad \text{implies} \]

\[ \mathcal{A}_0 \vdash \{ P \} \quad \text{check} \quad e \quad \text{end} \quad \{ (P \land e), (P \land \neg e) \} \]

Assume \( \models \{ P \} \quad \text{check} \quad e \quad \text{end} \quad \{ Q'_n, Q'_e \} \). For \( Q'_n \triangleq (P \land e) \) and \( Q'_e \triangleq (P \land \neg e) \), the conclusion follows by applying the rule of consequence, and the check axiom.

\[ \square \]

B.3.5 Loop Rule

We have to prove:

\[ \models \{ I \} \quad \text{until} \quad e \quad \text{loop} \quad s_1 \quad \text{end} \quad \{ (I \land e), R_e \} \quad \text{implies} \]

\[ \mathcal{A}_0 \vdash \{ I \} \quad \text{until} \quad e \quad \text{loop} \quad s_1 \quad \text{end} \quad \{ (I \land e), R_e \} \]
using the hypotheses

\[ \models \{ \neg e \land I \} \quad s_2 \{ \, I \land R_e \} \tag{implies} \quad A_0 \models \{ \neg e \land I \} \quad s_2 \{ \, I \land R_e \} \]

Assume \( \models \{ \, I \} \) until e loop \( s_1 \) end \( \{ \, T_n \land T_e \} \)

Let \( \{ P_0^n, R_e' \} \) be the strongest postcondition \( \{ P_0^n, P_0^e \} \triangleq s_1(P) \). Let \( \{ P_{n+1}^e, P_{e+1}^i \} \) be the strongest postcondition \( \{ P_{n+1}^e, P_{e+1}^i \} \triangleq s_2(P_n^i) \). Let \( I' \) be the invariant \( I' \triangleq \cup_i P_n^i \) and \( R_e' \triangleq \cup_i P_e^i \). Then by induction hypotheses we get:

\[ A_0 \models \{ \neg e \land I' \} \quad s_2 \{ \, I' \land R_e' \} \]

Finally, since \( I' \triangleq \cup_i P_n^i \) and \( R_e' \triangleq \cup_i P_e^i \) then \( I' \Rightarrow I \) and \( R_e' \Rightarrow R_e \). Then applying the loop rule and the rule of consequence we prove:

\[ A_0 \models \{ \, I \} \quad \text{until e loop} \ s_1 \quad \text{end} \quad \{ \, (I \land e) \land R_e \} \]

B.3.6 Read Attribute Axiom

We have to prove:

\[ \models \begin{cases} \ (y \neq \text{Void} \land P[\text{instvar}(y, S@a)/x]) \lor \\ \ (y = \text{Void} \land Q_e) \end{cases} x := y.S@a \quad \{ \, P \land Q_e \} \tag{implies} \]

\[ A_0 \models \begin{cases} \ (y \neq \text{Void} \land P[\text{instvar}(y, S@a)/x]) \lor \\ \ (y = \text{Void} \land Q_e) \end{cases} x := y.S@a \quad \{ \, P \land Q_e \} \]

Let \( P' \) be \( P' \triangleq (y \neq \text{Void} \land P[\text{instvar}(y, S@a)/x]) \lor (y = \text{Void} \land Q_e) \).

Assume that \( \models \{ \, P' \} \quad x := y.S@a \quad \{ \, Q'_n \land Q'_e \} \) holds, then

\[ \models P' \Rightarrow Q'_e[\text{instvar}(y, S@a)/x] \] Then, by the read attribute axiom and the consequence rule, we prove:

\[ A_0 \models \{ \, P' \} \quad x := y.S@a \quad \{ \, Q'_n \land Q'_e \} \]

\[ \square \]

B.3.7 Write Attribute Axiom

We have to prove:
B.3 Completeness Proof

We have to prove:

\[
\begin{array}{c}
\{ y \neq \text{Void} \land P[\$\text{instvar}(y, S@a) := e]/\$] \lor (y = \text{Void} \land Q_e) \} \\
y.S@a := e \quad \{ P, Q_e \}
\end{array}
\]

implies

\[
\mathcal{A}_0 \vdash \left\{ y.S@a := e \;\{ P, Q_e \} \right. \\
\{ (y \neq \text{Void} \land P[\$\text{instvar}(y, S@a) := e]/\$]) \lor (y = \text{Void} \land Q_e) \} \\
y.S@a := e \quad \{ P, Q_e \}
\]

Let \( P' \) be \( P' \triangleq (y \neq \text{Void} \land P[\$\text{instvar}(y, S@a) := e]/\$]) \lor (y = \text{Void} \land Q_e). \)

Assume \( \models \{ P' \} \quad y.S@a := e \quad \{ Q'_n, Q'_e \} \) holds, then

\( \models P' \Rightarrow Q'_n[\$\text{instvar}(y, S@a) := e]/\$]. \) Then, by the write attribute axiom and the consequence rule, we prove:

\[
\mathcal{A}_0 \vdash \{ P' \} \quad y.S@a := e \quad \{ Q'_n, Q'_e \}
\]

\( \Box \)

B.3.8 Local Rule

We have to prove:

\[
\models \{ P \} \quad \text{local} \; v_1 : T_1; \ldots; v_n : T_n; \; s \;\{ Q_n, Q_e \} \quad \text{implies} \\
\mathcal{A}_0 \vdash \{ P \} \quad \text{local} \; v_1 : T_1; \ldots; v_n : T_n; \; s \;\{ Q_n, Q_e \}
\]

using the induction hypothesis:

\[
\models \{ P \land v_1 = \text{init}(T_1) \land \ldots \land v_n = \text{init}(T_n) \} \; s \;\{ Q_n, Q_e \} \quad \text{implies} \\
\mathcal{A}_0 \vdash \{ P \land v_1 = \text{init}(T_1) \land \ldots \land v_n = \text{init}(T_n) \} \; s \;\{ Q_n, Q_e \}
\]

Assume \( \models \{ P \} \quad \text{local} \; v_1 : T_1; \ldots; v_n : T_n; \; s \;\{ Q_n, Q_e \}, \) then

\[
\models \{ P \land v_1 = \text{init}(T_1) \land \ldots \land v_n = \text{init}(T_n) \} \; s \;\{ Q'_n, Q'_e \}
\]

where \( \{ Q'_n, Q'_e \} \) is the strongest postcondition:

\[
\{ Q'_n, Q'_e \} \triangleq s(P \land v_1 = \text{init}(T_1) \land \ldots \land v_n = \text{init}(T_n))
\]

By induction hypothesis we have:

\[
\mathcal{A}_0 \vdash \{ P \land v_1 = \text{init}(T_1) \land \ldots \land v_n = \text{init}(T_n) \} \; s \;\{ Q'_n, Q'_e \}
\]

Since \( \{ Q'_n, Q'_e \} \) is the strongest postcondition, then \( Q_n \Rightarrow Q'_n \) and \( Q_e \Rightarrow Q'_e \). Then, by the consequence rule and the local rule, we prove:
\begin{align*}
\mathcal{A}_0 \models \{ P \} & \quad \text{local} \quad v_1 : T_1; \ldots \quad v_n : T_n; \quad s \quad \{ Q_n, Q_e \} \\
\square
\end{align*}

### B.3.9 Rescue Rule

Figure [B.1](#) shows a diagram of the states produced by the execution of the `rescue` clause. The instruction is $s_1 \text{ rescue } s_2$. The arrow with label $s_1$ means that the execution of the instruction $s_1$ starting in the state $P_n^i$ terminates in the state $\{ P_n^i, P_e^i \}$ where $P_n^i$ is the postcondition after normal termination, and $P_e^i$ is the postcondition when $s_1$ triggers an exception. In a similar way, the execution of the instruction $s_2$ stating in the state $P_e^i$ terminates in the state $\{ Q_n^i, Q_e^i \}$ where $Q_n^i$ is the postcondition after normal termination, and $Q_e^i$ is the postcondition when $s_1$ triggers an exception. If $\text{Retry} = \text{true}$ then the postcondition $Q_n^i$ implies $P_n^i$. If $\text{Retry} = \text{false}$ then the postcondition $Q_n^i$ implies $Q_n^i \land \neg \text{Retry} \land Q_e^i$. Furthermore, $Q_e^i$ implies $Q_n^i \land \neg \text{Retry} \land Q_e^i$.

![Fig. B.1: Completeness proof](image)

We have to prove:

\[
\models \{ P \} \quad s_1 \text{ rescue } s_2 \quad \{ Q_n, Q_e \}
\]

implies

\[
\mathcal{A}_0 \models \{ P \} \quad s_1 \text{ rescue } s_2 \quad \{ Q_n, Q_e \}
\]

using the hypotheses
\( B.3 \) Completeness Proof

\[ \models \{ I_r \} \ s_1 \ \{ Q_n \ , \ Q_e \} \implies \mathcal{A}_0 \vdash \{ I_r \} \ s_1 \ \{ Q_n \ , \ Q_e \} \text{ and} \]

\[ \models \{ Q_e \} \ s_2 \ \{ \text{Retry} \implies I_r \land \neg \text{Retry} \implies R_e \ , \ R_e \} \implies \mathcal{A}_0 \vdash \{ Q_e \} \ s_2 \ \{ \text{Retry} \implies I_r \land \neg \text{Retry} \implies R_e \ , \ R_e \} \text{ and} \]

\[ P \implies I_r \]

Assume \( \models \{ P \} \ s_1 \text{ rescue } s_2 \ \{ Q_n \ , \ Q_e \} \). Let

\[
P_n^0 \triangleq P \quad (B.21)
\]

\[
\{ P_n^i , P_e^i \} \triangleq s_1( P_n^i ) \quad (B.22)
\]

\[
\{ Q_n^i , Q_e^i \} \triangleq s_2( P_e^i ) \quad (B.23)
\]

\[
P_n^{i+1} \triangleq Q_n^i \land \text{Retry} \quad (B.24)
\]

\[
I_r \triangleq \bigcup_i P_n^i \quad (B.25)
\]

\[
T_n \triangleq \bigcup_i P_n^i \quad (B.26)
\]

\[
T_e \triangleq \bigcup_i P_e^i \quad (B.27)
\]

\[
R_e \triangleq \bigcup_i ((Q_n^i \land \neg \text{Retry}) \lor Q_e^i) \quad (B.28)
\]

We have \( T_n \implies Q_n \) and \( R_e \implies Q_e \). Then,

\[ \models \{ P_n^i \} \ s_1 \ \{ P_n^i , P_e^i \} , \text{ and} \]

\[ \models \{ P_e^i \} \ s_2 \ \{ Q_n^i , Q_e^i \} , \text{ for all } i \]

Therefore,

\[ \models \{ \bigcup_i P_n^i \} \ s_1 \ \{ \bigcup_i P_n^i , \bigcup_i P_e^i \} \text{ and} \]

\[ \models \{ \bigcup_i P_e^i \} \ s_2 \ \{ \bigcup_i Q_n^i , \bigcup_i Q_e^i \} \]

Then by induction hypotheses

\[ \mathcal{A}_0 \vdash \{ \bigcup_i P_n^i \} \ s_1 \ \{ \bigcup_i P_n^i , \bigcup_i P_e^i \} \text{ and} \]

\[ \mathcal{A}_0 \vdash \{ \bigcup_i P_e^i \} \ s_2 \ \{ \bigcup_i Q_n^i , \bigcup_i Q_e^i \} \]

Since \( \bigcup_i P_n^i \implies P \), and \( \bigcup_i P_n^i \implies Q_n \), and \( \bigcup_i Q_e^i \implies Q_e \), and the rule of consequence, we get

\[ \mathcal{A}_0 \vdash \{ I_r \} \ s_1 \ \{ Q_n , T_e \} \quad (B.29) \]
By (B.24) we know \( Q^i_n \Rightarrow (\text{Retry} \Rightarrow P^i_n) \) and by (B.28) \( Q^i_n \Rightarrow (\neg \text{Retry} \Rightarrow R_e) \).

Then, since \( I_r = \bigcup_i P^i_n \) we get \( Q^i_n \Rightarrow (\text{Retry} \Rightarrow I_r) \) and since \( R_e \Rightarrow Q_e \) we get \( Q^i_n \Rightarrow (\neg \text{Retry} \Rightarrow Q_e) \).

Then, by (B.28) \( \bigcup_i Q^i_e \Rightarrow R_e, R_e \Rightarrow Q_e \) and the rule of consequence, we prove:

\[
\mathcal{A}_0 \vdash \{ T_e \} \quad \text{s}_2 \quad \{ (\text{Retry} \Rightarrow I_r \land \neg \text{Retry} \Rightarrow Q_e), Q_e \} \tag{B.30}
\]

To finish the proof, we need to prove \( P \Rightarrow I_r \). This holds because \( P = P^0_n \) and \( I_r = \bigcup_i P^i_n \). Then from [B.29] and [B.30] and applying the rescue rule we get:

\[
\mathcal{A}_0 \vdash \{ P \} \quad \text{s}_1 \text{ rescue } \text{s}_2 \quad \{ Q_n, Q_e \}
\]

\(\square\)

### B.3.10 Routine Implementation Rule

We have to prove:

\[
\models \{ P \} \quad T@m \quad \{ Q_n, Q_e \} \quad \text{implies} \quad \mathcal{A}_0 \vdash \{ P \} \quad T@m \quad \{ Q_n, Q_e \}
\]

Assume \( \models \{ P \} \quad T@m \quad \{ Q_n, Q_e \} \).

\[
\mathcal{A}_0 \vdash \{ \$ = \$' \} \quad T@m \quad \{ Q^T@m_n, Q^T@m_e \}
\]

\[
\mathcal{A}_0 \vdash \{ \$ = \$' \land P[\$'/\$] \} \quad T@m \quad \{ Q^T@m_n \land P[\$'/\$], Q^T@m_e \land P[\$'/\$] \}
\]

\[
\mathcal{A}_0 \vdash \{ \$ = \$' \land P[\$'/\$] \} \quad T@m \quad \{ Q_n, Q_e \}
\]

\[
\mathcal{A}_0 \vdash \{ P \} \quad T@m \quad \{ Q_n, Q_e \}
\]

where (B.31), (B.32) are the following implications:

\[
Q^T@m_n \land P[\$'/\$] \Rightarrow Q_n \tag{B.31}
\]

\[
Q^T@m_e \land P[\$'/\$] \Rightarrow Q_e \tag{B.32}
\]

### B.3.11 Routine Invocation Rule

We have to prove:

\[
\models \{ P \} \quad x := y.T:m(e) \quad \{ Q_n, Q_e \} \quad \text{implies} \quad \mathcal{A}_0 \vdash \{ P \} \quad x := y.T:m(e) \quad \{ Q_n, Q_e \}
\]

Assume \( \models \{ P \} \quad x := y.T:m(e) \quad \{ Q_n, Q_e \} \). Then, by definition of \( \models \) we obtain:
\(|= \{ \text{Result'}/\text{Result} \} \) \quad \text{Result} := y.\text{T}\cdot m(e) \quad \{ Q_n[\text{Result'}/\text{Result}, \text{Result}/x] \text{, } Q_e[\text{Result'}/\text{Result}] \}

Let \( P', Q'_n \) and \( Q'_e \) be the following pre and postconditions:

\[
P' \triangleq P \begin{bmatrix} \text{Result'}/\text{Result}, \text{Current'}/\text{Current}, \text{Current}/y \end{bmatrix} \wedge p = e
\]

\[
Q'_n \triangleq Q_n \begin{bmatrix} \text{Result'}/\text{Result}, \text{Result}/x, \text{Current'}/\text{Current}, \text{Current}/y \end{bmatrix}
\]

\[
Q'_e \triangleq Q_e \begin{bmatrix} \text{Result'}/\text{Result}, \text{Current'}/\text{Current}, \text{Current}/y \end{bmatrix}
\]

By definition of \(|=\), we get:

\(|= \{ P' \} \) \quad \text{Result} := \text{Current}.\text{T}\cdot m(p) \quad \{ Q'_n, Q'_e \}

Then, by definition of \(|=\), we obtain:

\(|= \{ P' \} \) \quad \text{T}\cdot m(p) \quad \{ Q'_n, Q'_e \}

Then, we obtain the following derivation:

\[
\begin{array}{c}
\mathcal{A}_0 \vdash \{ P' \} \quad \text{T}\cdot m(p) \quad \{ Q'_n, Q'_e \} \\
\mathcal{A}_0 \vdash \{ \text{Result'}/\text{Result} \} \quad x := y.\text{T}\cdot m(e) \quad \{ Q''_n, Q''_e \}
\end{array}
\]

invocation rule

where \( P'', Q''_n \), and \( Q''_e \) are defined as follows:

\[
P'' \triangleq \overset{y \neq \text{Void}}{\text{y}} \wedge P'[\text{y}/\text{Current}, e/p] \wedge y = \text{Void} \wedge Q'_e[\text{y}/\text{Current}]
\]

\[
Q''_n \triangleq Q'_n[\text{y}/\text{Current}, x/\text{Result}]
\]

\[
Q''_e \triangleq Q'_e[\text{y}/\text{Current}]
\]

Unfolding the definition of \( P' \), and \( Q'_e \) we obtain
\[ P'' \triangleq \begin{cases} y \neq \text{Void} \land P & \text{Result}' / \text{Result}, \text{Current}' / \text{Current}, \text{Current} / y, y / \text{Current}, \\ y = \text{Void} \land Q_e & \text{Current} / y, y / \text{Current}, \text{Result}' / \text{Result}, \text{Current}' / \text{Current} \end{cases} \land e = e \land \]
\[ \equiv \begin{cases} y \neq \text{Void} \land P & \text{Result}' / \text{Result}, \text{Current}' / \text{Current} \\ y = \text{Void} \land Q_e & \text{Result}' / \text{Result}, \text{Current}' / \text{Current} \end{cases} \]

Also, unfolding the definition of \( Q''_n \) and \( Q''_e \) we know:

\[ Q''_n \triangleq Q_n \begin{cases} \text{Result}' / \text{Result}, \text{Current}' / \text{Current}, \text{Current} / y, y / \text{Current}, \\ \text{Result} / x, x / \text{Result} \end{cases} \equiv Q_n \begin{cases} \text{Result}' / \text{Result}, \text{Current}' / \text{Current} \end{cases} \]

\[ Q''_e \triangleq Q_e \begin{cases} \text{Result}' / \text{Result}, \text{Current}' / \text{Current}, \text{Current} / y, y / \text{Current} \end{cases} \equiv Q_e \begin{cases} \text{Result}' / \text{Result}, \text{Current}' / \text{Current} \end{cases} \]

Thus, the only replacement used in \( P'', Q''_n \), and \( Q''_e \) is \( \text{Result}' / \text{Result} \) and \( \text{Current}' / \text{Current} \). Now applying the invoc_var_rule with \( \text{Result}' \) and \( \text{Current}' \) we obtain the following derivation:

\[ A_0 \vdash \{ P'' \} x := y.T:m(p) \{ Q''_n, Q''_e \} \]
\[ A_0 \vdash \{ (y \neq \text{Void} \land P) \lor y = \text{Void} \land Q_e \} x := y.T:m(p) \{ Q_n, Q_e \} \quad \text{invoc_var_rule} \]

Finally, since we know \( (P \land y = \text{Void}) \Rightarrow Q_e \) from the hypothesis, applying the rule of consequence we prove:

\[ A_0 \vdash \{ P \} x := y.T:m(e) \{ Q_n, Q_e \} \]
\[ \square \]
B.3 Completeness Proof

B.3.12 Virtual Routines

To prove this case, $T:m$, we use the following lemma:

**Lemma 11** (Subtypes).

$$\forall T' \leq T : |= \{ P \land \tau(\text{Current}) = T' \} \ x_{T:m} \ {Q_n, Q_e} \text{ then}$$

$$\vdash \{ P \land \tau(\text{Current}) = T' \} \ x_{T':m} \ {Q_n, Q_e}$$

**Proof.** The proof follows from the proof rule for

$$\vdash \{ P \land \tau(\text{Current}) = T' \} \ x_{T':m} \ {Q_n, Q_e}$$

and completeness for the routine implementation rule presented in Section B.3.10.

Now, we prove the case $T:m$ as follows. We know:

$$|= \{ P \} \ x_{T:m} \ {Q_n, Q_e}$$

Applying Lemma 11 to all descendants of $T$, and applying the subtype rule and the consequence rule get:

$$\vdash \{ P \} \ x_{T:m} \ {Q_n, Q_e}$$

$\Box$
Soundness and Completeness Proof of the Logic
APPENDIX C

SOUNDNESS PROOF OF THE EIFFEL PROOF-TRANSFORMING COMPILER

This section presents the soundness proof of the Eiffel proof-transforming compiler. This proof includes the soundness proof of the core proof-transforming compiler (described in Chapter 6), and the Eiffel-specific proof-transforming compiler (presented in Chapter 7). The most interesting parts of the proof are the proof for the translation of rescue rule and the proof for the translation of once routines.

Before presenting the proofs, we show the soundness theorems described in Sections 6.7 and 7.4. Then, we prove soundness for the routine translator. Finally, we present the soundness proof for the expression translator and the instruction translator.

This appendix is based on the technical report [95].

C.1 Theorems

Soundness of the Eiffel proof-transforming compiler has been stated using three soundness theorems: (1) soundness of the routine translation; (2) soundness of the instruction translation; and (3) soundness of the contract translator. These theorems are defined as follows (they are described in Section 6.7 and Section 7.4):

Theorem 3 (Section 6.7)

\[ A \vdash \{ P \} \quad m \quad \{ Q_n, Q_e \} \]

then

\[ \vdash \nabla_B \left( \frac{Tree_1}{A \vdash \{ P \} \quad m \quad \{ Q_n, Q_e \}} \right) \]
Theorem 4 (Section 6.7)

\[
\vdash \text{Tree}_1 \quad \land \\
\quad \vdash \text{A} \{ P \} \quad s_1 \{ Q_n, Q_e \} \\
(I_{\text{start}} \ldots I_{\text{end}}) = \nabla_S \left( \vdash \text{Tree}_1 \quad \land \quad \vdash \text{A} \{ P \} \quad s_1 \{ Q_n, Q_e \} \right) \\
(Q_n \Rightarrow E_{\text{end}+1}) \land \\
(Q_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV} \Rightarrow E_{\text{exc}}) \land \\
\Rightarrow \\
\forall l \in I_{\text{start}} \ldots I_{\text{end}} : \vdash \{ E_l \} I_l
\]

Theorem 5 (Section 7.4)

\[
\forall b : \text{BoolExp, t : TypeFunc, e : Expr, c : CallRoutine, p : Argument} : \\
(\text{wellF}_C b) \Rightarrow (\text{value}_C b h_1 h_2 s) = ((\nabla_C b) h_1 h_2 s) \land \\
(\text{wellF}_T t) \Rightarrow (\text{value}_T t h_1 h_2 s) = ((\nabla_T t) h_1 h_2 s) \land \\
(\text{wellF}_\text{Exp} e) \Rightarrow (\text{value}_\text{Exp} e h_1 h_2 s) = ((\nabla_\text{Exp} e) h_1 h_2 s) \land \\
(\text{wellF}_\text{Call} c) \Rightarrow (\text{value}_\text{Call} c h_1 h_2 s) = ((\nabla_\text{Call} c) h_1 h_2 s) \land \\
(\text{wellF}_\text{Arg} p) \Rightarrow (\text{value}_\text{Arg} p h_1 h_2 s) = ((\nabla_\text{Arg} p) h_1 h_2 s)
\]

The proofs of Theorem 3 and Theorem 4 run by induction on the structure of the derivation tree for \( \{ P \} \quad s_1 \{ Q_n, Q_e \} \). These proofs are presented in Section C.2 and Section C.3. The proof of Theorem 5 runs by induction on the syntactic structure of the expression. This proof has been formalized and proved in Isabelle. The proof of Theorem 5 can be found in our technical report [95].

C.2 Soundness Proof of the Routine Translator

Since the logic for Eiffel and the logic for CIL support the same routine rules and the same language-independent rules, the translation of these rules is straightforward. The soundness proof is also simple. In this section, we present the proof for the class rule and the conjunction rule. The proofs for subtype, routine implementation, and the language-independent rules are similar.

C.2.1 Class Rule

The translation of the class rule has been presented in Section 6.2.1 on page 109. Let \( T_{\text{imp}} \) and \( T_{\text{tm}} \) be the following proof trees:
C.2 Soundness Proof of the Routine Translator

$$T_{imp} \equiv A \vdash \{ \tau(\text{Current}) = T \land P \} \ \text{imp}(T, m) \ \{ Q_n, Q_e \}$$

$$T_{tm} \equiv A \vdash \{ \tau(\text{Current}) \prec T \land P \} \ \text{T}:m \ \{ Q_n, Q_e \}$$

where $Tree_1$ and $Tree_2$ are the derivations used to prove the Hoare triples of $\text{imp}(T, m)$ and $T:m$ respectively.

We have to prove:

$$A \vdash \{ \tau(\text{Current}) \preceq T \land P \}$$

implies

$$\Gamma \vdash B( \text{imp} )$$

$$\Gamma \vdash B( \text{tm} )$$

using the induction hypotheses:

$$T_{imp} \text{ implies } \Gamma \vdash B( \text{imp} )$$

$$T_{tm} \text{ implies } \Gamma \vdash B( \text{tm} )$$

Since the application of the class rule in the source is a valid proof, then we know $T_{imp}$ and $T_{tm}$ are also valid. Then, we can apply the induction hypotheses and we get:

$$\Gamma \vdash B( \text{imp} )$$

$$\Gamma \vdash B( \text{tm} )$$

Applying the class rule in the CIL logic using the above CIL proofs, we conclude:

$$\Gamma \vdash \{ \tau(\text{Current}) \preceq T \land P \}$$

$$\{ Q_n, Q_e \}$$

$\square$
C.2.2 Conjunction Rule

The translation of the conjunction rule is described in Section 6.2.4 on page 113. Let $T_{S_1}$ and $T_{S_2}$ be the following proof trees:

$T_{S_1} \equiv \frac{\text{Tree}_1}{\text{A} \vdash \{ P' \} \quad s_1 \quad \{ Q'^n, Q'^e \}}$

$T_{S_2} \equiv \frac{\text{Tree}_2}{\text{A} \vdash \{ P'' \} \quad s_1 \quad \{ Q''^n, Q''^e \}}$

We have to prove:

$$
\frac{\text{A} \vdash \{ P' \land P'' \} \quad s_1 \quad \{ Q'^n \land Q''^n, Q'^e \land Q''^e \}}{\text{cil conjunction rule}}
$$

using the induction hypotheses:

$$
T_{S_1} \text{ implies } \nabla_B (T_{S_1})
$$

$$
T_{S_2} \text{ implies } \nabla_B (T_{S_2})
$$

Since the application of the conjunction rule in the source logic is valid, then applying the induction hypotheses we get:

$$
\nabla_B (T_{S_1})
$$

$$
\nabla_B (T_{S_2})
$$

Using the conjunction rule in the CIL logic, we show:

$$
\frac{\nabla_B (T_{S_1}) \quad \nabla_B (T_{S_2})}{\text{cil conjunction rule}}
$$

\[ \square \]

C.3 Soundness Proof of the Expression Translator

Before presenting the proof of the instruction translator, we prove soundness of the expression translation. Soundness of the expression translator is stated with the following lemma:
C.3 Soundness Proof of the Expression Translator

Lemma 12 (Soundness of Expression Translator).

\[
(I_{l_{\text{start}}}, \ldots I_{l_{\text{end}}}) = \nabla_E (Q \land \text{unshift}(P[c/s(0)]), \ e, \ shift(Q) \land P, \ l_{\text{start}}) \land (shift(Q) \land P \Rightarrow E_{l_{\text{end}}+1}) \Rightarrow \forall l \in l_{\text{start}} \ldots l_{\text{end}} : \vdash \{E_l\} I_l
\]

The proof of the expression translator runs by induction on the syntactic structure of the expression. Following, we present the proof for constants, variables, binary expressions and unary expressions.

C.3.1 Constants

The translation of constants is presented in Section 6.3.1 on page 116. We have to prove:

\[
I_{l_{\text{start}}} = \nabla_E (Q \land \text{unshift}(P[c/s(0)]), \ c, \ shift(Q) \land P, \ l_{\text{start}}) \land (shift(Q) \land P \Rightarrow E_{l_{\text{start}}+1}) \Rightarrow \vdash \{Q \land \text{unshift}(P[c/s(0)])) \}_{l_{\text{start}}} : \text{ldc } c
\]

To prove that this bytecode specification is valid, we have to show that the precondition \( \{Q \land \text{unshift}(P[c/s(0)])) \) implies the weakest precondition of \( \text{ldc } c \). To show this implication, first we prove:

\[
Q \land \text{unshift}(P[c/s(0)]) \text{ implies } wp(\text{ldc } c, shift(Q) \land P)
\]

then, by the hypothesis we know \( (shift(Q) \land P \Rightarrow E_{l_{\text{start}}+1}) \), and we prove

\[
Q \land \text{unshift}(P[c/s(0)]) \text{ implies } wp(\text{ldc } c, E_{l_{\text{start}}+1})
\]

The implication holds as follows:

\[
Q \land \text{unshift}(P[c/s(0)]) \text{ implies } wp(\text{ldc } c, shift(Q) \land P) \\
[\text{definition of } wp] \\
Q \land \text{unshift}(P[c/s(0)]) \text{ implies } unshift(\ shift(Q) \land P[c/s(0)] ) \\
[\text{definition of } unshift] \\
Q \land \text{unshift}(P[c/s(0)]) \text{ implies } Q \land \text{unshift}(P[c/s(0)])
\]

\( \Box \)
C.3.2 Variables

The translation of variables is described in Section 6.3.2 on page 116. We have to show:

\[ I_{l_{\text{start}}} = \nabla_E (Q \land \text{unshift}(P[x/s(0)]), \ x, \ shift(Q) \land P, \ l_{\text{start}}) \land \]
\[ (shift(Q) \land P \Rightarrow E_{l_{\text{start}}+1}) \]
\[ \Rightarrow \]
\[ \vdash \{ Q \land \text{unshift}(P[x/s(0)]) \} _{l_{\text{start}}} : \text{ldc} \ x \]

To show that the bytecode specification for \text{ldc} \ x is valid, we prove:

\[ Q \land \text{unshift}(P[x/s(0)]) \implies \text{wp}(\text{ldc} \ x, \text{shift}(Q) \land P) \]

then, by the hypothesis we have \( shift(Q) \land P \Rightarrow E_{l_{\text{start}}+1} \).

The proof is as follows:

\[ Q \land \text{unshift}(P[x/s(0)]) \implies \text{wp}(\text{ldc} \ x, \text{shift}(Q) \land P) \]
\[ \text{[definition of \ wp]} \]
\[ Q \land \text{unshift}(P[x/s(0)]) \implies \text{unshift}( shift(Q) \land P[x/s(0)] ) \]
\[ \text{[definition of \ unshift]} \]
\[ Q \land \text{unshift}(P[x/s(0)]) \implies Q \land \text{unshift}(P[x/s(0)]) \]
\[ \Box \]

C.3.3 Binary Expressions

Binary expressions has been translated in Section 6.3.3 on page 116. We have to prove:

\[ I_{l_{\text{start}}...l_c} = \nabla_E (Q \land \text{unshift}(P[e_1 \ op \ e_2/s(0)]), \ e_1 \ op \ e_2, \ shift(Q) \land P, \ l_{\text{start}}) \land \]
\[ (shift(Q) \land P \Rightarrow E_{l_{c+1}}) \]
\[ \Rightarrow \]
\[ \forall \ l \in l_{\text{start}}...l_c \vdash \{ E_l \} : \ I_l \]

using the hypotheses:
C.3 Soundness Proof of the Expression Translator

\[ l_{\text{start}} \ldots l_{\text{end}} = \]
\[ \nabla_E ( Q \land \text{unshift}(P[e_1 \text{ op } e_2/s(0)]) , e_1 , shift(Q) \land P[s(0) \text{ op } e_2/s(0)] , l_{\text{start}}) \land (shift(Q) \land P[s(0) \text{ op } e_2/s(0)] \Rightarrow E_{l_{\text{end}}}) \]
\[ \Rightarrow \forall l \in l_{\text{start}} \ldots l_{\text{end}} \vdash \{E_l\} : I_l \]

and
\[ l_{2\text{start}} \ldots l_{2\text{end}} = \]
\[ \nabla_E ( shift(Q) \land P[s(0) \text{ op } e_2/s(0)] , e_2 , shift^2(Q) \land shift P[s(1) \text{ op } s(0)/s(1)] , l_{2\text{start}}) \land (shift(Q) \land P[s(0) \text{ op } e_2/s(0)] \Rightarrow E_{l_{2\text{end}}}) \]
\[ \Rightarrow \forall l \in l_{2\text{start}} \ldots l_{2\text{end}} \vdash \{E_l\} : I_l \]

Since the precondition of \( l_c \) is equal to the postcondition of the translation of the expression \( e_2 \), we can apply the second induction hypothesis and we prove that the translation of the expression \( e_2 \) is valid. In a similar way, since the precondition of the translation of \( e_2 \) is equal to the postcondition of the translation of \( e_1 \), we can apply the first induction hypothesis, and we show that the translation of \( e_1 \) is valid. Finally, we can prove that the instruction specification at label \( l_c \) is valid applying the definition of \( \text{wp} \).

C.3.4 Unary Expressions

The translation of unary expressions is described in Section 6.3.4 on page 117. We have to prove:

\[ I_{l_{\text{start}} \ldots l_b} = \nabla_E ( Q \land \text{unshift}(P[\text{unop } e/s(0)]) , \text{unop } e , shift(Q) \land P , l_{\text{start}}) \land (shift(Q) \land P \Rightarrow E_{l_{b+1}}) \]
\[ \Rightarrow \forall l \in l_{\text{start}} \ldots l_b \vdash \{E_l\} : I_l \]

using the hypothesis:

\[ l_{1\text{start}} \ldots l_{1\text{end}} = \]
\[ \nabla_E ( Q \land \text{unshift}(P[\text{unop } e/s(0)]) , e , shift(Q) \land P[\text{unop } s(0)/s(0)] , l_{1\text{start}}) \land (shift(Q) \land P[\text{unop } s(0)/s(0)] \Rightarrow E_{l_{1\text{end}}}) \]
\[ \Rightarrow \forall l \in l_{1\text{start}} \ldots l_{1\text{end}} \vdash \{E_l\} : I_l \]
To apply the induction hypothesis, we need to show:

\[(\text{shift}(Q) \land P[\text{unop } s(0)/s(0)] \Rightarrow E_{l_1 end})\]

Since \(E_{l_1 end}\) is equal to \(E_{l_6}\), and \(E_{l_6}\) is defined as:

\[\text{shift}(Q) \land P[\text{unop } s(0)/s(0)]\]

then, the implication holds, and by induction hypothesis we get:

\[\forall l \in l_{1\text{start}}...l_{1\text{end}} : \vdash \{E_l\} I_l\]

Finally, we need to show that the instruction specification at \(l_b\) holds. By definition of \(wp\), we need to show:

\[(\text{shift}(Q) \land P[\text{unop } s(0)/s(0)] \text{ implies } wp(\text{unop}_{op}, E_{l_{b+1}}))\]

By hypothesis \((\text{shift}(Q) \land P \Rightarrow E_{l_{b+1}})\), applying the definition of \(wp\), we prove that

\[(\text{shift}(Q) \land P[\text{unop } s(0)/s(0)] \text{ implies } wp(\text{unop}_{op}, (\text{shift}(Q) \land P))\]

\[\Box\]

### C.4 Soundness Proof of the Instruction Translator

The soundness proof of the instruction translator runs by induction on the structure of the derivation tree for \(A \vdash \{ P \} s \{ Q_n, Q_e \}\). In this section, we present the proof for the most interesting cases.

#### C.4.1 Assignment Axiom

The assignment axiom is translated in Section 6.4.1 on page 118. Let \(P'\) be a precondition defined as follows:

\[P' \equiv (\text{safe}(e) \land P[e/x]) \lor (\neg\text{safe}(e) \land Q_e)\]

We have to prove:
C.4 Soundness Proof of the Instruction Translator

245

\[ \vdash A \vdash \{ P' \} x := e \{ P, Q_e \} \] 

\[ (I_{l_1} \ldots I_{l_b}) = \nabla_S \left( A \vdash \{ P' \} x := e \{ P, Q_e \}, l_a, l_{b+1}, l_{exc} \right) \] 

\[ (P \Rightarrow E_{l_{b+1}}) \wedge 
\]

\[ (Q_e \wedge excV \neq null \wedge s(0) = excV) \Rightarrow E_{l_{exc}} \wedge \]

\[ \forall l \in l_a \ldots l_b : \vdash \{ E_l \} I_l \]

Applying Lemma 12 we show that the translation of the expression \( e \) produces a valid bytecode proof. The bytecode specification for \texttt{stloc x} can be proved as follows:

\[ shift(safe(e) \wedge P[e/x]) \wedge s(0) = e \implies wp(stloc x, P) \]

\[ \text{[definition of wp]} \]

\[ shift(safe(e) \wedge P[e/x]) \wedge s(0) = e \implies shift(P[s(0)/x]) \]

\[ \square \]

C.4.2 Compound Rule

The translation of compound rule is described in Section 6.4.2 on page 119. Let \( T_{S_1} \) and \( T_{S_2} \) be the following proof trees:

\[ T_{S_1} \equiv \begin{array}{c} Tree_1 \\ \{ P \} s_1 \{ Q_n, R_e \} \end{array} \]

\[ T_{S_2} \equiv \begin{array}{c} Tree_2 \\ \{ Q_n \} s_2 \{ R_n, R_e \} \end{array} \]

We have to prove:

\[ \vdash A \vdash \{ P \} T_{S_1}, T_{S_2} \{ R_n, R_e \} \]

\[ (I_{l_1} \ldots I_{l_b}) = \nabla_S \left( A \vdash \{ P \} s_{1, \ldots, 2} \{ R_n, R_e \}, T_{S_1}, T_{S_2}, l_a, l_{b+1}, l_{exc} \right) \]

\[ (R_n \Rightarrow E_{l_{b+1}}) \wedge 
\]

\[ (R_e \wedge excV \neq null \wedge s(0) = excV) \Rightarrow E_{l_{exc}} \]

\[ \Rightarrow 
\]

\[ \forall l \in l_a \ldots l_b : \vdash \{ E_l \} I_l \]
By the first induction hypothesis we get:

\[
\begin{align*}
\vdash & \quad \text{Tree}_1 \\
& \quad A \vdash \{ P \} \quad s_1 \{ Q_n \, , \, R_e \} \\
(I_{l_a \ldots I_{l_a\text{-end}}}) &= \nabla_S \left( A \vdash \{ P \} \, s_1 \{ Q_n \, , \, R_e \} , \, l_a, l_b, l_{\text{exc}} \right) \\
& \quad (Q_n \Rightarrow E_{l_b\text{-end}+1}) \\
& \quad (R_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{l_{\text{exc}}} \\
& \quad \Rightarrow \forall l \in l_a \ldots l_{a\text{-end}} : \vdash \{ E_l \} I_l
\end{align*}
\]

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

\[
\begin{align*}
Q_n & \Rightarrow E_{l_b\text{-end}+1} \quad \text{(C.1)} \\
(R_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) & \Rightarrow E_{l_{\text{exc}}} \quad \text{(C.2)}
\end{align*}
\]

The second implication is proven by hypothesis. The precondition \(E_{l_b\text{-end}+1}\) is equal to \(E_{l_b}\). After the translation, the label precondition at \(l_b\) is \(Q_n\). Thus, \(Q_n \Rightarrow Q_n\), and we prove the first implication. Then, we can apply the induction hypothesis and get:

\[
\forall l \in l_a \ldots l_{a\text{-end}} : \vdash \{ E_l \} I_l
\]

The second induction hypothesis is:

\[
\begin{align*}
\vdash & \quad \text{Tree}_2 \\
& \quad A \vdash \{ Q_n \} \quad s_2 \{ R_n \, , \, R_e \} \\
(I_{l_b \ldots I_{l_b\text{-end}}}) &= \nabla_S \left( A \vdash \{ Q_n \} \, s_2 \{ R_n \, , \, R_e \} , \, l_b, l_{b+1}, l_{\text{exc}} \right) \\
& \quad (R_n \Rightarrow E_{l_{b+1}}) \\
& \quad (R_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{l_{\text{exc}}} \\
& \quad \Rightarrow \forall l \in l_b \ldots l_{b\text{-end}} : \vdash \{ E_l \} I_l
\end{align*}
\]

To apply the second induction hypothesis we have to show:

\[
\begin{align*}
Q_n & \Rightarrow E_{l_{b+1}} \quad \text{(C.3)} \\
(R_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) & \Rightarrow E_{l_{\text{exc}}} \quad \text{(C.4)}
\end{align*}
\]

These implications hold by hypothesis. Thus, we get:

\[
\forall l \in l_b \ldots l_{b\text{-end}} : \vdash \{ E_l \} I_l
\]
C.4 Soundness Proof of the Instruction Translator

Finally, we join both results and we get:

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

\[ \square \]

C.4.3 Conditional Rule

The conditional rule translation has been presented in Section 6.4.3 on page 119. Let $T_{S_1}$ and $T_{S_2}$ be the following proof trees:

$$T_{S_1} \equiv \begin{array}{c} \text{Tree}_1 \\ \mathcal{A} \vdash \{ P \land e \} \quad s_1 \quad \{ Q_n, Q_e \} \end{array}$$

$$T_{S_2} \equiv \begin{array}{c} \text{Tree}_2 \\ \mathcal{A} \vdash \{ P \land \neg e \} \quad s_2 \quad \{ Q_n, Q_e \} \end{array}$$

We have to show:

$$\vdash \begin{array}{c} T_{S_1} \quad T_{S_2} \\ \mathcal{A} \vdash \{ P \} \quad \text{if } e \text{ then } s_1 \quad \text{else } s_2 \quad \text{end} \quad \{ Q_n, Q_e \} \end{array} \quad \wedge$$

$$\left( I_{l_a} \ldots I_{l_e} \right) = \nabla_S \left( \begin{array}{c} T_{S_1} \quad T_{S_2} \\ \mathcal{A} \vdash \{ P \} \quad \text{if } e \text{ then } s_1 \quad \text{else } s_2 \quad \text{end} \quad \{ Q_n, Q_e \} \end{array} , \ l_a, l_{e+1}, l_{\text{exc}} \right) \quad \wedge$$

$$\left( Q_n \Rightarrow E_{l_{e+1}} \right) \quad \wedge$$

$$\left( ( Q_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV} ) \Rightarrow E_{l_{\text{exc}}} \right)$$

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

Applying Lemma 12 we prove that the translation of the expression $e$ is valid. The instruction specification at label $l_b$ holds applying the definition of $wp$. The specification at label $l_d$ holds by hypothesis ($Q_n \Rightarrow E_{l_{e+1}}$).

The proof of the translation for $s_1$ is as follows. By the first induction hypothesis we get:
\[\vdash Tree_1 \quad \wedge \quad A \vdash \{ P \} \quad s_1 \quad \{ Q_n, Q_e \} \quad \wedge \quad (I_{l_1} \ldots I_{l_{c,\text{end}}}) = \nabla_S \left( A \vdash \{ P \} \quad s_1 \quad \{ Q_n, Q_e \}, \quad l_c, l_d, l_{\text{exc}} \right) \wedge \]

\[(Q_n \Rightarrow E_{l_d}) \quad \wedge \quad (R_e \wedge excV \neq \text{null} \wedge s(0) = excV) \Rightarrow E_{l_{\text{exc}}} \Rightarrow \forall l \in l_c \ldots l_{c,\text{end}} : \vdash \{ E_l \} I_l \]

Since \( E_{l_d} \) is equal to \( Q_n \), and \((R_e \wedge excV \neq \text{null} \wedge s(0) = excV) \Rightarrow E_{l_{\text{exc}}} \) holds by hypothesis, we can apply the induction hypothesis, and we prove:

\[\forall l \in l_c \ldots l_{c,\text{end}} : \vdash \{ E_l \} I_l \]

The proof of the translation of \( s_2 \) is similar to the translation of \( s_1 \). Thus, we have proven:

\[\forall l \in l_a \ldots l_c : \vdash \{ E_l \} I_l \]

\[\square\]

### C.4.4 Check Axiom

The check axiom is translated in Section 6.4.4 on page 121. This proof is straight forward, applying the definition of \( wp \). We have to prove:

\[\vdash A \vdash \{ P \} \quad \text{check} \quad e \quad \text{end} \quad \{ (P \wedge e), (P \wedge \neg e) \} \quad \wedge \quad (I_{l_1} \ldots I_{l_d}) = \nabla_S \left( A \vdash \{ P \} \quad \text{check} \quad e \quad \text{end} \quad \{ (P \wedge e), (P \wedge \neg e) \}, \quad l_a, l_{d+1}, l_{\text{exc}} \right) \wedge \]

\[((P \wedge \neg e) \Rightarrow E_{l_{d+1}}) \quad \wedge \quad ((P \wedge e \wedge excV \neq \text{null} \wedge s(0) = excV) \Rightarrow E_{l_{\text{exc}}} \Rightarrow \forall l \in l_a \ldots l_d : \vdash \{ E_l \} I_l \]

The translation of the expression \( e \) holds by Lemma 12. To prove that the instruction at \( l_b \) holds, we have to show:

\[shift(P) \wedge s(0) = e \implies wp(brtrue \ l_{d+1})\]
Applying the definition of \( wp \), we have to prove:

\[
(shift(P) \land s(0) = e) \Rightarrow (\neg s(0) \Rightarrow shift(E_{l_c})) \land (s(0) \Rightarrow shift(E_{l_d+1}))
\]

The first implication holds because \( shift(E_{l_c}) = P \land \neg e \). The second implication holds by the hypothesis: \( (P \land e) \Rightarrow E_{l_d+1} \). Then using the definition of \( wp \) we prove the implication.

Applying the definition of \( wp \), we show that the instructions specifications at \( l_c \) and \( l_d \) are valid. Then, joining the proof we have:

\[
\forall l \in l_a \ldots l_d : \vdash \{ E_l \} I_l
\]

\[\square\]

### C.4.5 Loop Rule

The translation of loop rule is developed in Section 6.4.5 on page 121. Let \( T_{S_1} \) be the following proof tree:

\[
T_{S_1} \equiv \frac{\text{Tree}_1}{\mathcal{A} \vdash \{ \neg e \land I \} s_1 \{ I \land e \land R_e \}}
\]

We have to prove:

\[
\vdash \mathcal{A} \vdash \{ I \} \text{ until } e \text{ loop } s_1 \{ I \land e \land R_e \} \land (I \land e) \Rightarrow E_{l_d+1} \land (R_e \land excV \neq null \land s(0) = excV) \Rightarrow E_{l_{exc}}
\]

\[\Rightarrow \forall l \in l_a \ldots l_d : \vdash \{ E_l \} I_l\]

Since the translation of the expression \( e \) establishes \( I \), then the instruction specification at \( l_a \) is valid. The bytecode produced by the translation of \( e \) is valid (applying Lemma 12). Furthermore, the bytecode specification at \( l_d \) holds by definition of \( wp \) and the hypothesis \( (I \land e) \Rightarrow E_{l_d+1} \).

Now, we need to show that the translation of \( s_1 \) is valid. By the first induction hypothesis we get:
C.4.6 Read and Write Attribute Rule

The proofs are straightforward applying the definition of wp.

C.4.7 Routine Invocation Rule, Local Rule, and Creation Rule

These rules are proven applying the definition of weakest precondition.

C.4.8 Rescue Rule

The translation of the rescue rule is described in Section 7.2.3 on page 148. Let $T_{S_1}$ and $T_{S_2}$ be the following proof trees:

$$ T_{S_1} \equiv \begin{array}{c} \{ I_r \} \\ s_1 \{ Q_n, Q_e \} \end{array} \quad T_{S_2} \equiv \begin{array}{c} \{ Q_e \} \\ s_2 \{ \text{Retry} \Rightarrow I_r \land \neg\text{Retry} \Rightarrow R_e, R_e \} \end{array} $$

We have to prove:
We first prove that the translations of the instructions $s_1$ and $s_2$ produce valid bytecode proofs, and then we show that the specification of the remaining instructions hold. By the first induction hypothesis we get:

\[
\vdash s_1 \text{rescue} s_2 \left\{ \begin{array}{l} Q_n, R_e \end{array} \right\}, l_a, l_{i+1}, l_{exc} \quad \land \\
\vdash \left\{ \begin{array}{l} P \end{array} \right\}, s_1 \text{rescue} s_2 \left\{ \begin{array}{l} Q_n, R_e \end{array} \right\}, l_a, l_{i+1}, l_{exc} \quad \land \\
(\forall l \in l_a \ldots l_i : \vdash \{ E_l \} I_l)
\]

Due to the precondition $E_{l_b}$ is $Q_n$ and the precondition $E_{l_e}$ is:

\[
( Q_e \land \text{exc}V \neq \text{null} \land s(0) = \text{exc}V ) \quad \Rightarrow \quad E_{l_e}
\]

we can apply the first induction hypothesis and get:

\[
( Q_e \land \text{exc}V \neq \text{null} \land s(0) = \text{exc}V ) \quad \Rightarrow \quad E_{l_e}
\]

The second induction hypothesis is:
\[ \begin{align*}
&\vdash \text{Tree}_2 \\
&\{Q_e\} \ s_2 \begin{cases} 
\text{Retry} \Rightarrow I_r \land R_e \\
\neg\text{Retry} \Rightarrow R_e \land R_e 
\end{cases} \\
&(I_d ... I_{d,end}) = \nabla s \begin{cases} 
\text{Tree}_2 \\
\{Q_e\} \ s_2 \begin{cases} 
\text{Retry} \Rightarrow I_r \land R_e \\
\neg\text{Retry} \Rightarrow R_e \land R_e 
\end{cases} , l_d, l_e, l_{exc} \end{cases} \land \\
&(\ (\text{Retry} \Rightarrow I_r \land \neg\text{Retry} \Rightarrow R_e) \Rightarrow E_{l_e}) \land \\
&(\ (R_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{l_{exc}}) \\
\Rightarrow \\
\forall \ l \in l_d ... l_{d,end} : \vdash \{E_l\} I_l
\end{align*} \]

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

\[ \begin{align*}
&(\ (\text{Retry} \Rightarrow I_r \land \neg\text{Retry} \Rightarrow R_e) \Rightarrow E_{l_e}) \\
&(R_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{l_{exc}}
\end{align*} \]

The first implication holds because the precondition \( E_{l_e} \) is:

\[ (\text{Retry} \Rightarrow I_r \land \neg\text{Retry} \Rightarrow R_e) \]

The second implication holds by the hypothesis. So, we can apply the second induction hypothesis and get:

\[ \forall \ l \in l_d ... l_{d,end} : \vdash \{E_l\} I_l \]

The specification

\[ \{Q_n\} \ l_b : \text{leave} \ l_{i+1} \]

holds by the definition of \( \text{wp} \) for \text{leave}, and the hypothesis \( Q_n \Rightarrow E_{l_{i+1}} \).

Finally, the specifications at labels \( l_c, l_e, l_f, l_g, l_h, \) and \( l_i \) hold by the definition of \( \text{wp} \). For example, the instruction specification at label \( l_f \) is proven as follows:
C.4 Soundness Proof of the Instruction Translator

(Retry ⇒ Iᵣ ∧ ¬Retry ⇒ Rₑ ∧ s(0) = Retry) implies wp (brfalse lᵢ, Eᵢ)
[definition of wp]

(Retry ⇒ Iᵣ ∧ ¬Retry ⇒ Rₑ ∧ s(0) = Retry) implies
( s(0) ⇒ shift(Eᵢ) ∧ ¬s(0) ⇒ shift(Eᵢ))
[definition of Eᵢ and Eᵢ]

(Retry ⇒ Iᵣ ∧ ¬Retry ⇒ Rₑ ∧ s(0) = Retry) implies
( s(0) ⇒ shift(Iᵣ) ∧ ¬s(0) ⇒ shift(Rₑ))
[definition of shift, and Iᵣ and Rₑ do not refer to the stack]

(Retry ⇒ Iᵣ ∧ ¬Retry ⇒ Rₑ ∧ s(0) = Retry) implies ( s(0) ⇒ Iᵣ ∧ ¬s(0) ⇒ Rₑ)
[this holds since s(0) = Retry]

Joining the proofs we get:

∀ l ∈ lₐ ... lₗ : ⊢ {Eᵢₗ} Iᵢₗ

□

C.4.9 Once functions Rule

The translation of once functions is described in Section 7.2.3 on page 149. Let P be the following precondition, where T_M_RES is a logical variable:

\[
P \equiv \begin{cases} 
- T@m\_done \land P' \lor 
T@m\_done \land P'' \land T@m\_result = T_M\_RES \land \neg T@m\_exc \lor 
T@m\_done \land P''' \land T@m\_exc 
\end{cases}
\]

and let Qₙ' and Qₑ' be the following postconditions:

\[
Qₙ' \equiv \begin{cases} 
T@m\_done \land \neg T@m\_exc \land 
(Qₙ \lor (P'' \land Result = T_M\_RES \land T@m\_result = T_M\_RES )) 
\end{cases}
\]

\[
Qₑ' \equiv \begin{cases} 
T@m\_done \land T@m\_exc \land (Qₑ \lor P''') 
\end{cases}
\]

Let T_body be the following proof tree:
Soundness Proof of the Eiffel Proof-Transforming Compiler

\[ T_{\text{body}} \equiv \frac{\mathcal{A}, \{P\} \quad T@m \ \{Q'_n, \ Q'_e\} \vdash}{\text{Tree}_1} \]

\[
\{ P'[\text{false/T@m\_done}] \land T@m\_done \} \quad \text{body}(T@m) \quad \{ Q_n, \ Q_e \}
\]

We have to prove:

\[
\vdash_{\text{T}_{\text{body}}} \mathcal{A} \vdash \{ P \} \quad T@m \ \{ Q'_n, \ Q'_e \} \land (I_{l_a} \ldots I_{l_r}) = \nabla_S \left( \mathcal{A} \vdash \{ P \} \quad T@m \ \{ Q'_n, \ Q'_e \}, \ l_a, l_{r+1}, l_{exc} \right) \land \left( Q'_n \Rightarrow E_{l_{r+1}} \right) \land \left( (Q'_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV} \Rightarrow E_{l_{exc}} \right) \Rightarrow \forall l \in l_a \ldots l_r : \vdash \{ E_l \} \ l_l
\]

The instruction specifications at labels \( l_a, \ldots, l_d, l_f, \ldots, l_r \) hold by the definition of \( wp \). For example, we prove the instruction specification at \( l_a \) as follows:

\[
P \text{ implies } wp (\text{ldsfld T@m\_done, shift(P) \land s(0) = T@m\_done})
\]

[definition of \( wp \)]

\[
P \text{ implies } \text{unshift( shift(P) \land s(0) = T@m\_done[T@m/s(0)] )}
\]

[definition of replacement]

\[
P \text{ implies } \text{unshift( shift(P) \land T@m = T@m\_done)}
\]

[definition of unshift and shift]

\[
P \text{ implies } (P \land T@m = T@m\_done)
\]

\( \square \)

The interesting case of the proof is the translation of \( T_{\text{body}} \). By the induction hypothesis we get:
\[ \vdash T_{\text{body}} \land (I_i \ldots I_{\text{le}}) = \nabla_S (T_{\text{body}}, I_e, I_f, I_s) \land (Q_n \Rightarrow E_{\ell}) \land ((\text{shift}(Q_e) \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{\ell}) \]

\[ \Rightarrow \forall l \in I_e \ldots I_{\text{le}} : \vdash \{E_l\} I_l \]

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

\[ (Q_n \Rightarrow E_{\ell}) \]

\[ ((\text{shift}(Q_e) \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{\ell}) \]

The precondition at label $E_{\ell}$ is $Q_n$, so the first implication holds. The precondition $E_{\ell}$ is:

\[ (\text{shift}(Q_e) \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \]

then the second implication also holds, and we can apply the induction hypothesis and get:

\[ \forall l \in I_e \ldots I_{\text{le}} : \vdash \{E_l\} I_l \]

Finally, joining the proofs we get:

\[ \forall l \in I_e \ldots I_e : \vdash \{E_l\} I_l \]

\[ \square \]
Soundness Proof of the Eiffel Proof-Transforming Compiler
Chapter 8 describes a Java proof-transforming compiler using both CIL and Java Bytecode. The translation using CIL is simpler due to CIL supports try-catch and try-finally instructions. This chapter presents the soundness proofs for the CIL proof translation and the Java Bytecode proof translation. The proofs run by induction on the structure of the derivation tree for

\[ \{P\} s_1 \{Q_n, Q_b, Q_e\} \]

Section D.1 presents the soundness proof of the CIL proof-transforming compiler. In Section D.2 we prove soundness for the Java Bytecode PTC. The proof of the CIL PTC is also simpler than the proof of the Java Bytecode PTC due to the use of try-catch and try-finally instructions.

This appendix is partially based on the technical report [81].

D.1 Soundness of the Proof Translator using CIL

This section describes the soundness proof of the CIL proof translator. We first present the theorem, and then we prove soundness for compound, while, break, throw, try-catch, and try-finally rules.

D.1.1 Theorem

The soundness theorem for the CIL proof translator adds an extra hypothesis to relate the break postcondition with the precondition at the label \( l_{break} \). Furthermore, it uses the mapping function \( m \) for the exception labels. The theorem is the following: (this theorem has been described in Section 8.1.8 on page 161)
Theorem 6 (Section 8.1.8)

\[ \vdash \text{Tree}_1 \{ \{ P \}, s_1 \{ \{ Q_n; Q_b; Q_e \} \} \wedge \]

\[ (I_{\text{start}} \ldots I_{\text{end}}) = \nabla_S \left( \text{Tree}_1 \{ \{ P \}, s_1 \{ \{ Q_n; Q_b; Q_e \} \}, l_{\text{start}}, l_{\text{end}+1}, l_{\text{break}}, m \right) \wedge \]

\[ (Q_n \Rightarrow E_{l_{\text{end}+1}}) \wedge \]

\[ (Q_b \Rightarrow E_{l_{\text{break}}}) \wedge \]

\[ (\forall T : \text{Type} : T \preceq \text{Throwable} : (Q_e \wedge \text{exc} \neq \text{null} \wedge s(0) = \text{exc} V) \Rightarrow E_{m[T]} \) \]

\[ \Rightarrow \forall l \in l_{\text{start}} \ldots l_{\text{end}} : \vdash \{ E_l \} I_l \]

D.1.2 Compound Rule

The translation of the compound rule is described in Section 8.1.2 on page 156. Let \( T_{S_1} \) and \( T_{S_2} \) be the following proof trees:

\[ T_{S_1} \equiv \begin{array}{c}
\vdash \\
\{ P \} \\
s_1 \{ Q_n; R_b; R_e \} \\
\end{array} \]

\[ T_{S_2} \equiv \begin{array}{c}
\vdash \\
\{ Q_n \} \\
s_2 \{ R_n; R_b; R_e \} \\
\end{array} \]

Replacing the compound rule in Theorem 6, we have to prove:

\[ \vdash \begin{array}{c}
T_{S_1} \\
T_{S_2} \\
\end{array} \{ P \} \\
\{ s_1; s_2 \} \\
\{ R_n; R_b; R_e \} \\
\end{array} \wedge 

\[ (I_{l_a} \ldots I_{l_b}) = \nabla_S \left( \begin{array}{c}
T_{S_1} \\
T_{S_2} \\
\{ P \} \\
\{ s_1; s_2 \} \\
\{ R_n; R_b; R_e \} \\
\end{array} \\
\begin{array}{c}
l_{a}; l_{b+1}, l_{\text{break}}, l_{\text{exc}}, m \end{array} \right) \wedge 

\[ (R_n \Rightarrow E_{l_{b+1}}) \wedge \]

\[ (R_b \Rightarrow E_{l_{\text{break}}}) \wedge \]

\[ (\forall T : \text{Type} : T \preceq \text{Throwable} : (R_e \wedge \text{exc} \neq \text{null} \wedge s(0) = \text{exc} V) \Rightarrow E_{m[T]} \) \]

\[ \Rightarrow \forall l \in l_a \ldots l_b : \vdash \{ E_l \} I_l \]

By the first induction hypothesis we get:
D.1 Soundness of the Proof Translator using CIL

\[
\begin{align*}
\vdash & \quad \text{Tree}_1 \quad \wedge \\
& \quad \{ P \} \quad s_1 \quad \{ Q_n, R_b, R_e \} \\
& (I_a \ldots I_{a_{\text{end}}}) = \nabla_S \left( \text{tree}_1 \right) \quad \{ P \} \quad s_1 \quad \{ Q_n, R_b, R_e \} \quad , \quad l_a, l_b, l_{\text{break}}, l_{\text{exc}}, m \quad \wedge \\
& (Q_n \Rightarrow E_{l_b}) \quad \wedge \\
& (R_b \Rightarrow E_{l_{\text{break}}}) \quad \wedge \\
& (\forall T : \text{Type} : T \preceq \text{Throwable} : (R_e \land \text{exc} \not= \text{null} \land s(0) = \text{exc}) \Rightarrow E_{m[T]} ) \\
\Rightarrow \\
& \forall l \in l_a \ldots l_{a_{\text{end}}} : \vdash \{ E_l \} \ I_l
\end{align*}
\]

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

\[
\begin{align*}
Q_n & \Rightarrow E_{l_b} \quad \text{(D.1)} \\
R_b & \Rightarrow E_{l_{\text{break}}} \quad \text{(D.2)} \\
(\forall T : \text{Type} : T \preceq \text{Throwable} : (R_e \land \text{exc} \not= \text{null} \land s(0) = \text{exc}) \Rightarrow E_{m[T]} ) \\
\Rightarrow \\
& \forall l \in l_a \ldots l_{a_{\text{end}}} : \vdash \{ E_l \} \ I_l
\end{align*}
\]

The second and the third implications are proven by hypothesis since the break label \(l_{\text{break}}\) and the mapping function \(m\) are not modified by the translation. The precondition \(E_{l_b}\) is \(Q_n\). Thus, \(Q_n \Rightarrow Q_n\) holds. Then, we can apply the first induction hypothesis and get:

\[
\forall l \in l_a \ldots l_{a_{\text{end}}} : \vdash \{ E_l \} \ I_l
\]

The second induction hypothesis is:

\[
\begin{align*}
\vdash & \quad \text{Tree}_2 \quad \wedge \\
& \quad \{ Q_n \} \quad s_2 \quad \{ R_n, R_b, R_e \} \\
& (I_b \ldots I_{b_{\text{end}}}) = \nabla_S \left( \text{tree}_2 \right) \quad \{ Q_n \} \quad s_2 \quad \{ R_n, R_b, R_e \} \quad , \quad l_b, l_{b+1}, l_{\text{break}}, l_{\text{exc}}, m \quad \wedge \\
& (R_n \Rightarrow E_{l_{b+1}}) \quad \wedge \\
& (R_b \Rightarrow E_{l_{\text{break}}}) \quad \wedge \\
& (\forall T : \text{Type} : T \preceq \text{Throwable} : (R_e \land \text{exc} \not= \text{null} \land s(0) = \text{exc}) \Rightarrow E_{m[T]} ) \\
\Rightarrow \\
& \forall l \in l_b \ldots l_{b_{\text{end}}} : \vdash \{ E_l \} \ I_l
\end{align*}
\]
To apply this hypothesis, we need to show:

\[ R_n \Rightarrow E_{b+1} \quad (D.4) \]

\[ R_b \Rightarrow E_{break} \quad (D.5) \]

\[ \forall T : Type : T \leq Throwable : \left( R_e \land excV \neq \text{null} \land s(0) = excV \right) \Rightarrow E_{m[T]} \quad (D.6) \]

These implications holds by hypothesis. So, we get:

\[ \forall l \in l_b \ldots l_{\text{end}} : \vdash \{ E_l \} I_l \]

Finally, we join both results and we prove:

\[ \forall l \in l_a \ldots l_b : \vdash \{ E_l \} I_l \]

\[ \square \]

### D.1.3 While Rule

The while rule is translated using CIL in Section 8.1.3 on page 157. Let \( T_{S_1} \) and \( T_{\text{while}} \) be the following proof trees:

\[
T_{S_1} = \frac{T_{\text{Tree1}}}{e \land I} \quad s_1 \{ I, Q_b, R_e \}
\]

\[
T_{\text{while}} = \frac{T_{S_1}}{I} \quad \text{while} (e) s_1 \{ (I \land \neg e) \lor Q_b, false, R_e \}
\]

We have to show:

\[ \vdash T_{\text{while}} \land \]

\[ (I_a \ldots I_d) = \nabla_S (T_{\text{while}}, l_a, l_{d+1}, l_{\text{break}}, l_{\text{exc}}, m) \land \]

\[ (I \land \neg e) \lor Q_b \Rightarrow E_{l_{d+1}} \land \]

\[ (false \Rightarrow E_{l_{\text{break}}}) \land \]

\[ (\forall T : Type : T \leq Throwable : (R_e \land excV \neq \text{null} \land s(0) = excV) \Rightarrow E_{m[T]}) \Rightarrow \]

\[ \forall l \in l_a \ldots l_d : \vdash \{ E_l \} I_l \]
The instruction specifications at labels $l_a$ and $l_b$ hold by definition of $wp$ and the hypothesis. The translation of the expression $e$ holds by Lemma 12. To prove that the translation of the instruction $s_1$ produces a valid CIL proof, we use the following induction hypothesis:

$$
\vdash T_{s_1} \land (I_{l_b \ldots l_{b,\text{end}}}) = \nabla_S (T_{s_1}, l_b, l_c, l_{d+1}, l_{\text{exc}}, m) \land (I \Rightarrow E_{l_b}) \land (Q_b \Rightarrow E_{l_{d+1}}) \land (\forall T : Type : T \leq \text{Throwable} : (R_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{m[T]})
$$

$$
\Rightarrow \forall l \in l_b \ldots l_{b,\text{end}} : \vdash \{E_l\} I_l
$$

To apply this hypothesis, we need to show:

$$
I \Rightarrow E_{l_e} \quad \text{(D.7)}
$$

$$
Q_b \Rightarrow E_{l_{d+1}} \quad \text{(D.8)}
$$

$$
\forall T : Type : T \leq \text{Throwable} : (R_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{m[T]} \quad \text{(D.9)}
$$

The first implication holds since $E_{l_e}$ is equal to $I$. The second implication holds from the hypothesis

$$(I \land \neg e) \lor Q_b \Rightarrow E_{l_{d+1}}$$

And the third implication is true by hypothesis. Then, applying the induction hypothesis we get:

$$
\forall l \in l_b \ldots l_{b,\text{end}} : \vdash \{E_l\} I_l
$$

Joining the proofs, we show:

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

\[\square\]

### D.1.4 Break Rule

The break rule has been translated in Section 8.1.4 on page 158. The goal to prove is:
\[ \vdash \{ P \} \break \{ \text{false, P, false} \} \land \\
( I_\text{start} = \nabla_S \left( \begin{array}{c}
\{ P \} \break \{ \text{false, P, false} \}, l_\text{start}, l_\text{start+1}, l_\text{break}, l_\text{exc}, m
\end{array} \right) \land \\
( \text{false} \Rightarrow E_{l_\text{start+1}} ) \land \\
( P \Rightarrow E_{l_\text{break}} ) \land \\
( \forall T : \text{Type} : T \leq \text{Throwable} : ( \text{false} \land \text{exc} \neq \text{null} \land s(0) = \text{exc} ) \Rightarrow E_{m[T]} ) \Rightarrow \\
\{ P \} \ l_\text{start} : \text{br} \ l_\text{break} \]

By definition of wp, we have to show:

\[ P \Rightarrow E_{l_\text{break}} \]

The implication holds by hypothesis.

\[ \square \]

### D.1.5 Throw Rule

The translation of this rule is presented in Section 8.1.5 on page 158.

We have to prove:

\[ \vdash \{ P[e/\text{exc}V] \} \ \text{throw} \ e \ \{ \text{false, false, P} \} \land \\
( I_a...I_b = \nabla_S \left( \begin{array}{c}
\{ P[e/\text{exc}V] \} \ \text{throw} \ e \ \{ \text{false, false, P} \}, l_a, l_{b+1}, l_\text{break}, \ l_\text{exc}, m
\end{array} \right) \land \\
( \text{false} \Rightarrow E_{l_{b+1}} ) \land \\
( \text{false} \Rightarrow E_{l_\text{break}} ) \land \\
( \forall T : \text{Type} : T \leq \text{Throwable} : ( P \land \text{exc} \neq \text{null} \land s(0) = \text{exc} ) \Rightarrow E_{m[T]} ) \Rightarrow \\
\forall l \in l_a ... l_b : \vdash \{ E_l \} \ I_l \]

The translation of the expression \( e \) holds by Lemma 12. To show that the bytecode specification at label \( l_b \) is valid, one needs to show that its precondition implies the
precondition at the label where the exception is caught. This precondition is $E_{m[T]}$, then we need to show:

$$\text{shift}(P[e/exV]) \land s(0) = e \implies \text{wp(throw, } E_{m[T]}\})$$

This implication holds by the hypothesis:

$$\forall T : Type : T \preceq Throwable : (P \land exc \neq \text{null} \land s(0) = exc) \implies E_{m[T]}$$

□

### D.1.6 Try-catch Rule

The translation of the try-catch rule using CIL is described in Section 8.1.6 on page 159. Let $T_{S_1}$ and $T_{S_2}$ be the following proof trees:

$$T_{S_1} \equiv \begin{array}{c} Tree_1 \\ \{ P \} \\ s_1 \{ Q_n, Q_b, Q \} \end{array}$$

$$T_{S_2} \equiv \begin{array}{c} Tree_2 \\ \{ Q_e[e/exV] \} \\ s_2 \{ Q_n, Q_b, R_e \} \end{array}$$

where

$$Q \equiv ( (Q''_e \land \tau(excV) \nleq T) \lor (Q'_e \land \tau(excV) \nleq T) )$$

Let $R$ be the following postcondition:

$$R \equiv (R_e \lor (Q''_e \land \tau(excV) \nleq T) )$$

We need to show:
Soundness Proof of the Java Proof-Transforming Compiler

\[ \vdash T_{S_1} \quad T_{S_2} \quad \land \]

\[ \{ P \} \quad \text{try } s_1 \quad \text{catch } (T \ e) \quad s_2 \quad \{ Q_n, Q_b, R \} \]

\[ (I_{a_1} \ldots I_{a_n}) = \nabla_S \left( T_{S_1} \quad T_{S_2} \quad \land \right) \]

\[ \{ P \} \quad \text{try } s_1 \quad \text{catch } (T \ e) \quad s_2 \quad \{ Q_n, Q_b, R \} \quad l_a, l_{e+1}, l_{\text{break}}, l_{\text{exc}}, m \] \land

\[ (Q_n \Rightarrow E_{l_{e+1}}) \quad \land \]

\[ (Q_b \Rightarrow E_{l_{\text{break}}}) \quad \land \]

\[ (\forall T : \text{Type} : T \leq \text{Throwable} : (R \land \text{exc}\  \neq \  \text{null} \land s(0) = \text{exc}\) \Rightarrow E_{m[T]}) \]

\[ \Rightarrow \forall l \in l_a \ldots l_e : \vdash \{ E_l \} \quad I_l \]

Applying the first induction hypothesis we get:

\[ \vdash \quad \text{Tree}_1 \quad \land \]

\[ \{ P \} \quad s_1 \quad \{ Q_n, Q_b, Q \} \quad l_a, l_b, l_{\text{break}}, l_{\text{exc}}, m' \] \land

\[ (Q_n \Rightarrow E_{l_b}) \quad \land \]

\[ (Q_b \Rightarrow E_{l_{\text{break}}}) \quad \land \]

\[ (\forall T : \text{Type} : T \leq \text{Throwable} : (Q \land \text{exc}\  \neq \  \text{null} \land s(0) = \text{exc}\) \Rightarrow E_{m'[T]}) \]

\[ \Rightarrow \forall l \in l_a \ldots l_{a_{\text{end}}} : \vdash \{ E_l \} \quad I_l \]

where \( m' \) is defined as \( m[T \rightarrow l_c] \)

To apply the induction hypothesis, we have to show:

\[ Q_n \Rightarrow E_{l_b} \ (\text{D.10}) \]

\[ Q_b \Rightarrow E_{l_{\text{break}}} \ (\text{D.11}) \]

\[ \forall T : \text{Type} : T \leq \text{Throwable} : (Q \land \text{exc}\  \neq \  \text{null} \land s(0) = \text{exc}\) \Rightarrow E_{m'[T]} \ (\text{D.12}) \]

The first two implications holds by hypothesis and the definition of \( E_{l_b} \). In the third implication, \( m' \) is defined as \( m[T \rightarrow l_c] \). Then the implication holds for all exception types except for \( T \) from the hypothesis:

\[ \forall T : \text{Type} : T \leq \text{Throwable} : (R \land \text{exc}\  \neq \  \text{null} \land s(0) = \text{exc}\) \Rightarrow E_{m[T]} \]
In the case of $T$, $m'[T]$ is $E_l$. By the definition of $E_l$, we need to prove:

$$(Q \land excV \neq null \land s(0) = excV) \Rightarrow \left( \text{shift}(Q'_e) \land excV \neq null \land \tau(excV) \preceq T \land s(0) = excV \right)$$

Since the exception type is $T$, then $\tau(excV) \preceq T$. Then by definition of $Q$ we have:

$$(Q'_e \land \tau(excV) \preceq T \land excV \neq null \land s(0) = excV) \Rightarrow \left( \text{shift}(Q'_e) \land excV \neq null \land \tau(excV) \preceq T \land s(0) = excV \right)$$

By definition of $\text{shift}$, $\text{shift}(Q'_e) = Q'_e$ because $Q'_e$ does not refer to the stack. Then, the implication holds, and we get by the first induction hypothesis:

$$\forall l \in l_d \ldots l_{d,\text{end}} : \vdash \{E_i\} I_l$$

The instructions at $l_b$, $l_c$, and $l_e$ are proven by hypothesis and the definition of $wp$. The second induction hypothesis is:

$$\vdash \text{Tree}_2 \left\{ Q'_e[e/excV] \right\} s_2 \left\{ Q_n, Q_b, R_e \right\} \land (\forall T : \text{Type} : T \preceq \text{Throwable} : (R_e \land excV \neq null \land s(0) = excV) \Rightarrow E_m[T])$$

$$\Rightarrow \forall l \in l_d \ldots l_{d,\text{end}} : \vdash \{E_i\} I_l$$

To apply this induction hypothesis, we need to show that the following implications hold:

$$Q_n \Rightarrow E_l \ (D.13)$$

$$Q_b \Rightarrow E_{\text{break}} \ (D.14)$$

$$\forall T : \text{Type} : T \preceq \text{Throwable} : (R_e \land excV \neq null \land s(0) = excV) \Rightarrow E_m[T] \ (D.15)$$

Since the precondition $E_l$ is $Q_n$, the first implication holds. The second and the third implication hold by hypothesis. Then we get

$$\forall l \in l_d \ldots l_{d,\text{end}} : \vdash \{E_i\} I_l$$

Joining the proofs we show:

$$\forall l \in l_a \ldots l_e : \vdash \{E_i\} I_l$$

$\square$
D.1.7 Try-finally Rule

The proof of the try-finally rule is similar to the try-catch rule.

D.2 Soundness of the Proof Translator using Java Bytecode

This section proves the soundness theorem for the Java proof-transforming compiler using Java Bytecode. The proof is more complex than the proof presented in the above section due to the generation of the exception tables. First, we show the soundness theorem described in Section 8.2.6. Then, we prove Lemmas 3 and 4, which define the properties of the generated exception tables. These lemmas allow proving soundness of the proof transformation. Finally, we present the proofs for compound, while, try-finally, and break rules.

D.2.1 Theorem

The soundness theorem presented in Section 8.2.6 is the following:

**Theorem 7 (Section 8.2.6)**

\[
\begin{align*}
\vdash \{ P \} \quad & s_1 \{ \begin{array}{l}
Q_n, Q_b, Q_e \\
\end{array} \} \triangleq T_{S_1} \land \\
[(l_{start}, \ldots, l_{end}), etl] = \nabla_S (T_{S_1}, l_{start}, l_{end+1}, l_{break}, f, etl') \land \\
(\forall T : Type : (T \leq Throwable \lor T \equiv any)) : \\
(Q_e \land \text{exc}V \neq \text{null} \land s(0) = \text{exc}V) \Rightarrow E_{etl[l_{start}, l_{end}, T]} \land \\
(Q_n \Rightarrow E_{l_{end+1}}) \land \\
(Q_b \Rightarrow (f = \emptyset \Rightarrow (Q_b \Rightarrow E_{l_{break}}))) \land \\
f \neq \emptyset \Rightarrow \forall i \in 1..k : \\
(\vdash \{ U^i \} \quad s_i \{ V^i \} \land (V^i \Rightarrow U^{i+1}_b) \land \\
(\forall T : Type : ((T \leq Throwable) \lor (T \equiv any)) : \\
(V^i_e \land \text{exc}V \neq \text{null} \land s(0) = \text{exc}V) \Rightarrow E_{etl[T]} \land \\
(Q_b \Rightarrow U^i_b) \land (V_b^{k} \Rightarrow E_{l_{break}}) \land \\
(\forall etl = r^i_1 \cap r^i_2 \cap \ldots \cap r^i_m : (r^i_1 \cap r^i_2 \cap \ldots \cap r^i_m = \emptyset) \land (etl_i \subseteq etl')))
\end{align*}
\]

\[
\Rightarrow \forall l \in l_{start} \ldots l_{end} : \vdash \{ E_l \} \quad I_i
\]

We assume that the pre and postconditions \( U^i \) and \( V^i \) have a normal form defined by the pre and postconditions of the try-finally rule (described in Section 8.2.4). This
assumption is correct since the only translation that adds triples to the finally function \( f \) is the translation of try-finally rule.

### D.2.2 Proof of Lemmas 3 and 4

**Lemma 3 (Section 8.2.5)**

If

\[
\nabla_S \left( \begin{array}{c} \text{Tree} \\
\{ \{P\} \} \\
\{ \{Q_n, Q_b, Q_e\} \}
\end{array} \right), l_a, l_{b+1}, l_{\text{break}}, f, et \right) = \left((I_{l_a}...I_{l_b}), et'\right)
\]

and \( l_{\text{start}} \leq l_a < l_b \leq l_{\text{end}} \) then,

for every \( l_s, l_e : \text{Label} \) such that \( l_b < l_s < l_e \leq l_{\text{end}}, \) and

for every \( T : \text{Type} \) such that \( T \preceq \text{Throwable} \lor T \equiv \text{any}, \) the following holds:

\[et[l_{\text{start}}, l_{\text{end}}, T] = et'[l_s, l_e, T]\]

**Proof of Lemma 3 (Section 8.2.5)**

This proof is done by case analysis over the structure of the exception table \( et' \). Due to the exception table \( et' \) is produced by the translation \( \nabla_S \), it only can have the structure: \( et' = et + et_a \) where \( et_a \) is a new exception table created by \( \nabla_S \) or \( et' = \text{divide}(et, r, l'_a, l'_b) \) where \( et \) is an exception table, \( r \) is an exception entry stored in \( f \) and defined as \( r \triangleq [l_a, l_b, l_t, T] \), and \( l_a \leq l'_a < l'_b \leq l_b \). This property holds because \( \nabla_S \) only adds new lines to the exception table taken as parameter \( (et) \) in the translation of try-catch and try-finally, and \( \nabla_S \) only divides the table in the translation of break.

**Case \( et' = et + et_a \)**

The exception table \( et_a \) is produced by:

\[
\nabla_S \left( \begin{array}{c} \text{tree} \\
\{ \{P\} \} \\
\{ \{Q_n, Q_b, Q_e\} \}
\end{array} \right), l_a, l_{b+1}, l_{\text{break}}, f, et \right)
\]

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

Since we know:

\[\forall l_s, l_e : l_b < l_s < l_e \leq l_{\text{end}}\]

then \( l_{\text{start}} \leq l_s < l_e \leq l_{\text{end}} \) holds. Thus, searching for the target label between \( l_s \) and \( l_e \) (\( et[l_s, l_e, T] \)) is equivalent to searching for the target label between \( l_{\text{start}} \) and \( l_{\text{end}} \).
(\text{et}[\text{l}_{\text{start}}, \text{l}_{\text{end}}, \text{T}] ). \text{ Then } \text{et}[\text{l}_{\text{s}}, \text{l}_{\text{e}}, \text{T}] = \text{et}[\text{l}_{\text{start}}, \text{l}_{\text{end}}, \text{T}]. \text{ So, this proves that } \text{et}'[\text{l}_{\text{s}}, \text{l}_{\text{e}}, \text{T}] = \text{et}[\text{l}_{\text{start}}, \text{l}_{\text{end}}, \text{T}].

\textbf{Case } \text{et}' = \text{divide}(\text{et}, \text{r}, \text{l}'_{\text{a}}, \text{l}'_{\text{b}})

The definition of the function \text{divide} has been presented in Section 8.2.5 on page 167. There are two possible results after the execution of \text{divide}(\text{et}, \text{r}, \text{l}'_{\text{a}}, \text{l}'_{\text{b}}) either \text{divide} does not modify the exception entry \([\text{l}_{\text{start}'}, \text{l}_{\text{end}'}, \text{t}, \text{T}] \) in \text{et}[\text{l}_{\text{start}}, \text{l}_{\text{end}}, \text{T}] with \text{l}_{\text{start'}} \leq \text{l}_{\text{start}} and \text{l}_{\text{end'}} \leq \text{l}_{\text{end}}) or it does.

If \text{divide}(\text{et}, \text{r}, \text{l}'_{\text{a}}, \text{l}'_{\text{b}}) does not modify the exception entry \([\text{l}_{\text{start}'}, \text{l}_{\text{end}'}, \text{t}, \text{T}] \), then using a similar reasoning to the above case, we can conclude that \text{et}'[\text{l}_{\text{s}}, \text{l}_{\text{e}}, \text{T}] = \text{et}[\text{l}_{\text{start}}, \text{l}_{\text{end}}, \text{T}].

If \text{divide}(\text{et}, \text{r}, \text{l}'_{\text{a}}, \text{l}'_{\text{b}}) modifies the exception entry \([\text{l}_{\text{start}'}, \text{l}_{\text{end}'}, \text{t}, \text{T}] \) then it will be divided as:
\[
[\text{l}_{\text{start}'}, \text{l}_{\text{a}'}, \text{t}, \text{T}] + [\text{l}_{\text{b}'}, \text{l}_{\text{end}'}, \text{t}, \text{T}]
\]

We are looking for the target label between \text{l}_{\text{s}} and \text{l}_{\text{e}}. But we know \text{l}_{\text{b}} < \text{l}_{\text{s}} < \text{l}_{\text{e}} \leq \text{l}_{\text{end}} and \text{l}_{\text{a}} \leq \text{l}'_{\text{a}} < \text{l}'_{\text{b}} \leq \text{l}_{\text{b}}. \text{ Then } \text{et}'[\text{l}_{\text{s}}, \text{l}_{\text{e}}, \text{T}] \text{ will return the target label in the divided line } [\text{l}_{\text{b}'}, \text{l}_{\text{end'}}, \text{t}, \text{T}], \text{ which target label is the same than } \text{et}[\text{l}_{\text{start}}, \text{l}_{\text{end}}, \text{T}]. \text{ So, we can conclude } \text{et}[\text{l}_{\text{start}}, \text{l}_{\text{end}}, \text{T}] = \text{et}'[\text{l}_{\text{s}}, \text{l}_{\text{e}}, \text{T}].

\hfill \Box

\textbf{Lemma 4 (Section 8.2.5)} \text{ Let } r : \text{ExcTableEntry} \text{ and } \text{et}' : \text{ExcTable} \text{ be such that } r \in \text{et}'.

\text{If } \text{et} : \text{ExcTable} \text{ and } \text{l}_{\text{s}}, \text{l}_{\text{e}} : \text{Label} \text{ are such that } \text{et} = \text{divide}(\text{et}', \text{r}, \text{l}_{\text{s}}, \text{l}_{\text{e}}), \text{ then }

\text{et}[\text{l}_{\text{s}}, \text{l}_{\text{e}}, \text{T}] = r[\text{T}]

\textbf{Proof of Lemma 4 (Section 8.2.5)}

The proof runs by induction on \text{et}'.

\textit{Base case: } \text{et}' = [ ]

\text{By definition of } \text{divide}, \text{ divide}([ ], \text{r}, \text{l}_{\text{s}}, \text{l}_{\text{e}}) = [r]. \text{ Then the lemma holds.}

\textit{Inductive Case: } \text{divide}(r_{\text{i}} + \text{et}'_{\text{i}}, \text{r}, \text{l}_{\text{s}}, \text{l}_{\text{e}})

\text{If } r \nsubseteq r_{\text{i}} \text{ then the function } \text{divide} \text{ either updates } r_{\text{i}} \text{ or not depending on whether } (\text{r.from} > r_{\text{i}.to} \land \text{r.to} < r_{\text{i}.to}) \text{ holds or not. But, it does not update } r. \text{ Then by induction hypothesis we get } \text{divide}(r_{\text{i}} + \text{et}'_{\text{i}}, \text{r}, \text{l}_{\text{s}}, \text{l}_{\text{e}})[\text{l}_{\text{s}}, \text{l}_{\text{e}}, \text{T}] = r[\text{T}].

\text{If } r \subseteq r_{\text{i}} \text{ then the function } \text{divide} \text{ returns } r_{\text{i}} + \text{et}'_{\text{i}}. \text{ Then } r_{\text{i}} + \text{et}'_{\text{i}}[\text{l}_{\text{s}}, \text{l}_{\text{e}}, \text{T}] = r[\text{T}] \text{ because } r \subseteq r_{\text{i}} \text{ and } \text{divide} \text{ does not change } r_{\text{i}}.

\hfill \Box

Following, we present the soundness proof for compound, while, try-finally, and break rules.
D.2 Soundness of the Proof Translator using Java Bytecode

D.2.3 Compound Rule

The Java translation of the compound rule using Java Bytecode is described in Section 8.2.2 on page 164. Let \( T_{S_1} \) and \( T_{S_2} \) be the following proof trees:

\[
T_{S_1} \equiv \begin{array}{c}
\{ P \} \\
\{ Q_n, R_b, R_e \}
\end{array} \quad T_{S_2} \equiv \begin{array}{c}
\{ Q_n \} \\
\{ R_n, R_b, R_e \}
\end{array}
\]

Replacing the compound rule in Theorem 7, we have to prove:

\[
\vdash T_{S_1} T_{S_2} \begin{array}{c}
\{ P \} \\
\{ Q_n, R_b, R_e \}
\end{array} \quad \land \\
\land ((I_{l_0}...I_{l_b}), \ et_2) = \nabla S \left( T_{S_1} T_{S_2} \begin{array}{c}
\{ P \} \\
\{ Q_n, R_b, R_e \}
\end{array} , \ l_a, \ l_{b+1}, \ l_{\text{break}}, \ f, \ et \right) \land \\
(\forall T : \text{Type} : ((T \preceq \text{Throwable}) \lor (T \equiv \text{any})) : \quad \begin{array}{c}
(R_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{et[I_{l_a}...I_{l_b},T]} \end{array} \land \\
(R_n \Rightarrow E_{l_{b+1}}) \land \\
(R_b \Rightarrow (f = \emptyset \Rightarrow (R_b \Rightarrow E_{b_{\text{break}}})) \land \\
f \neq \emptyset \Rightarrow \forall i \in 1..k : \quad \left( \begin{array}{c}
(\vdash \{ U^i \} \ s_i \ \{ V^i \} \land (V_b^i \Rightarrow U_b^{i+1}) \land \\
(\forall T : \text{Type} : ((T \preceq \text{Throwable}) \lor (T \equiv \text{any})) : \\
(V_e^i \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{et,T} \end{array} \land \\
(R_b \Rightarrow U_b^1) \land (V_b^k \Rightarrow E_{b_{\text{break}}}) \land \\
(\forall et_i = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset) \land (et_i \subseteq et)) \right) \\
\Rightarrow \\
\forall l \in \ l_a \ldots \ l_b : \vdash \{ E_l \} I_l
\]

By the first induction hypothesis, we get:
\[
\vdash T_{S_1} \land \\
[(I_a...I_{a\text{end}}), et_1] = \nabla_S (T_{S_1}, l_a, l_b, l_{break}, f, et) \land \\
(\forall T : Type : ((T \preceq \text{Throwable}) \lor (T \equiv \text{any})) : \\
(R_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{et\{l_a, l_{a\text{end}}, T\}}) \land \\
(Q_n \Rightarrow E_{b}) \land \\
(R_b \Rightarrow (f = \emptyset \Rightarrow (R_b \Rightarrow E_{a\text{break}}))) \land \\
(f \neq \emptyset \Rightarrow \forall i \in 1..k : \\
(\vdash \{U_1\} s_i \{V_1\} \land (V_i^1 \Rightarrow U_i^{i+1}) \land \\
(\forall T : Type : ((T \preceq \text{Throwable}) \lor (T \equiv \text{any})) : \\
(V_i^i \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{et\{T\}}) \land \\
(R_b \Rightarrow U_b^1) \land (V_b^k \Rightarrow E_{b\text{break}}) \land \\
(\forall et_i = r_1^i + r_2^i + ... + r_m^i : (r_1^i \cap r_2^i \cap ... \cap r_m^i = \emptyset) \land (et_i \subseteq et))) \\
\Rightarrow \\
\forall l \in l_a...l_{a\text{end}} : \vdash \{E_l\} I_l
\]
\begin{align*}
\vdash T_{S_2} & \land \\
[(I_{b_0} \ldots I_{b_{\text{end}}}) \land \epsilon_t] = \nabla_{S_2} (T_{S_2}, \ b_b, \ l_{b+1}, \ l_{\text{break}}, \ f, \ \epsilon_t) & \land \\
(\forall \ T : \text{Type} : ((T \not\subseteq \text{Throwable}) \lor (T \equiv \text{any})) : \\
\quad (R_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{\text{et}_l[I_{b_0}, l_{b_{\text{end}}, T}]}) & \land \\
(R_n \Rightarrow E_{l_{b+1}}) & \land \\
(R_b \Rightarrow (f = \emptyset \Rightarrow (R_b \Rightarrow E_{\text{break}})) & \land \\
f \neq \emptyset \Rightarrow \forall \ i \in 1..k : \\
\quad (\vdash \{ U^i \} \subseteq \{ V^i \} \land (V^i_b \Rightarrow U^i_{b+1}) & \land \\
\quad (\forall \ T : \text{Type} : ((T \not\subseteq \text{Throwable}) \lor (T \equiv \text{any})) : \\
\quad \quad (V^i_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{\text{et}_l[T]} & \land \\
\quad \quad (R_b \Rightarrow U^i_b) \land (V^i_b \Rightarrow E_{\text{break}}) & \land \\
\quad \quad (\forall \epsilon_t = r^i_1 + r^i_2 + \ldots + r^m_i : (r^i_1 \cap r^i_2 \cap \ldots \cap r^m_i = \emptyset) \land (\epsilon_t \subseteq \epsilon_t))) & \\
\Rightarrow \\
\forall \ l \in l_b \ldots l_{b_{\text{end}}} : \vdash \{ E_l \} & I_l\end{align*}

The implication \( R_n \Rightarrow E_{l_{b+1}} \) holds by hypothesis. Furthermore, the implication

\begin{align*}
\forall \ T : \text{Type} : ((T \not\subseteq \text{Throwable}) \lor (T \equiv \text{any})) : \\
\quad (R_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{\text{et}_l[I_{b_0}, l_{b_{\text{end}}, T}]}
\end{align*}

holds by the hypothesis:

\begin{align*}
\forall \ T : \text{Type} : ((T \not\subseteq \text{Throwable}) \lor (T \equiv \text{any})) : \\
\quad (R_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{\text{et}_l[I_{b_0}, l_{b}, T]}
\end{align*}

and Lemma 3. The finally properties holds the hypothesis:

\begin{align*}
(R_b \Rightarrow (f = \emptyset \Rightarrow (R_b \Rightarrow E_{\text{break}})) & \land \\
f \neq \emptyset \Rightarrow \forall \ i \in 1..k : \\
\quad (\vdash \{ U^i \} \subseteq \{ V^i \} \land (V^i_b \Rightarrow U^i_{b+1}) & \land \\
\quad (\forall \ T : \text{Type} : ((T \not\subseteq \text{Throwable}) \lor (T \equiv \text{any})) : \\
\quad \quad (V^i_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{\text{et}_l[T]} & \land \\
\quad \quad (R_b \Rightarrow U^i_b) \land (V^i_b \Rightarrow E_{\text{break}}) & \land \\
\quad \quad (\forall \epsilon_t = r^i_1 + r^i_2 + \ldots + r^m_i : (r^i_1 \cap r^i_2 \cap \ldots \cap r^m_i = \emptyset) \land (\epsilon_t \subseteq \epsilon_t))) & \\
\end{align*}

and Lemma 3. Thus, we can apply the second induction hypothesis and prove:
∀ \ l \in l_b \ldots l_{b\text{end}}: \vdash \{ E_l \} I_l

Finally, we join both result and we get:

∀ \ l \in l_a \ldots l_b : \vdash \{ E_l \} I_l

\square

D.2.4 While Rule

The while rule is translated using Java Bytecode in Section 8.2.3 on page 165. Let $T_{S_1}$ be the following proof tree:

$$T_{S_1} \equiv \begin{array}{c}
\frac{T_{s_1}}{\{ e \land I \}} s_1 \{ I, Q_b, R_e \}
\end{array}$$

We have to prove:

$$\vdash T_{S_1} \begin{array}{c}
\{ I \} \\
\{ (I \land \neg e) \lor Q_b, false, R_e \}
\end{array}$$

$$\left[ (l_{a}, \ldots l_{d}), e_{t_1} \right\} =

\nabla S \left( \begin{array}{c}
\vdash \{ I \} \\
\{ (I \land \neg e) \lor Q_b, false, R_e \}
\end{array} \right) \land

(\forall T: Type : ((T \leq \text{Throwable}) \lor (T \equiv \text{any})) :

(R_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{et[l_a, l_d, T]} \land

(((I_n \land \neg e) \lor Q_b) \Rightarrow E_{l_{d+1}}) \land

(false \Rightarrow (f = \emptyset \Rightarrow (false \Rightarrow E_{break}))) \land

\left( f \neq \emptyset \Rightarrow \forall i \in 1..k:

\left( \begin{array}{c}
\vdash \{ U^i \} s_i \{ V^i \} \land (V^i \Rightarrow U^{i+1}_b) \land
\end{array} \right)

(\forall T: Type : ((T \leq \text{Throwable}) \lor (T \equiv \text{any})) :

(V^i_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{et_i[T]} \land

(false \Rightarrow U^i_b) \land (V^i_k \Rightarrow E_{\text{break}}) \land

(\forall et_i = r^i_1 + r^i_2 + \ldots + r^i_m : (r^i_1 \cap r^i_2 \cap \ldots \cap r^i_m = \emptyset) \land (et_i \subseteq et))\right)\right)

\Rightarrow \forall \ l \in l_a \ldots l_d : \vdash \{ E_l \} I_l
The instruction specification at labels \( l_a \) and \( l_d \) hold by definition of \( wp \). The translation of the expression \( e \) is valid by Lemma 12. The most interesting part of the proof is the translation of \( s_1 \). By the induction hypothesis, we get:

\[
\vdash T_{S_1} \land \\
[(I_{l_b} \ldots I_{l_{h\text{-end}}}), et_1] = \\
\nabla_s (T_{S_1}, l_b, l_c, l_{d+1}, f, et) \land \\
(\forall T : Type : ((T \leq \text{Throwable}) \lor (T \equiv \text{any})) : \\
(R_e \land \text{exc}V \neq \text{null} \land s(0) = \text{exc}V) \implies E_{et[l_b,l_{h\text{-end}},T]} ) \land \\
(I \implies E_{l_k}) \land \\
(Q_b \implies (f = \emptyset \implies (Q_b \implies E_{l_{d+1}})) \land \\
(f \neq \emptyset \implies \forall i \in 1..k : \\
\left( \\
\vdash \{U_i^i\} s_i \{V_i^i\} \land (V_b^i \implies U_b^{i+1}) \land \\
(\forall T : Type : ((T \leq \text{Throwable}) \lor (T \equiv \text{any})) : \\
(V_e^i \land \text{exc}V \neq \text{null} \land s(0) = \text{exc}V) \implies E_{et[T_i]} ) \land \\
(Q_b \implies U_b^i_k) \land (V_b^k \implies E_{\text{throw}_e}) \land \\
(\forall et_i = r_i^1 + r_i^2 + \ldots + r_i^m : (r_i^1 \cap r_i^2 \cap \ldots \cap r_i^m = \emptyset) \land (et \subseteq et)) \\
\right) \\
\Rightarrow \\
\forall l \in l_b \ldots l_{h\text{-end}} : \vdash \{E_l\} I_l
\]

Since the exception table \( et \) is the same than the theorem hypothesis, we know:

\[
(\forall T : Type : ((T \leq \text{Throwable}) \lor (T \equiv \text{any})) : \\
(R_e \land \text{exc}V \neq \text{null} \land s(0) = \text{exc}V) \implies E_{et[l_b,l_{h\text{-end}},T]} )
\]

Since the finally function \( f \) is \( \emptyset \), to be able to apply the induction hypothesis we have to show that

\[
Q_b \implies E_{l_{d+1}} \\
I \Rightarrow E_{l_k}
\]

(D.16)

(D.17)

From the theorem hypothesis we have \((I \land \neg e) \lor Q_b \) \implies \( E_{l_{d+1}} \). Then \( Q_b \implies E_{l_{d+1}} \) holds because \( Q_b \implies ((I_a \land \neg e) \lor Q_b) \) \implies \( E_{l_{d+1}} \). The second implication holds since the precondition \( E_{l_k} \) is \( I \). Now, we can apply the induction hypothesis and we get:
∀ \ l \in l_b \ldots l_{b, end} : \vdash \{ E_l \} \ I_l

Thus, we prove:

∀ \ l \in l_a \ldots l_d : \vdash \{ E_l \} \ I_l

\square

D.2.5 Try-finally Rule

The try-finally rule is translated using Java Bytecode in Section 8.2.4 on page 165.

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

\[
T_{S_1} \equiv \begin{array}{c}
\text{Tree}_1 \\
\{ P \} s_1 \{ Q_n, Q_b, Q_e \}
\end{array} \quad T_{S_2} \equiv \begin{array}{c}
\text{Tree}_2 \\
\{ Q \} s_2 \{ R, R'_b, R'_e \}
\end{array}
\]

where

\[
Q \equiv \begin{cases}
(Q_n \land X Tmp = \text{normal}) \lor (Q_b \land X Tmp = \text{break}) \lor \\
(Q_e[e Tmp/\text{exc V}] \land X Tmp = \text{exc} \land e Tmp = \text{exc V})
\end{cases}
\]

\[
R \equiv \begin{cases}
(R'_n \land X Tmp = \text{normal}) \lor (R'_b \land X Tmp = \text{break}) \lor \\
(R'_e[e Tmp/\text{exc V}] \land X Tmp = \text{exc})
\end{cases}
\]

We have to prove:
D.2 Soundness of the Proof Translator using Java Bytecode

\[
\vdash T_{S_1} T_{S_2} \quad \land
\]

\fbox{\begin{align*}
&\text{[(} l_a, ... l_b \text{), } et_4 \text{]} = \\
&\nabla_S \left( T_{S_1} T_{S_2} \quad \land
\begin{cases}
\{ P \} \text{ try } s_1 \text{ finally } s_2 \quad \{ R'_n, R'_b, R'_e \} \\
\forall T : \text{Type} : ((T \leq \text{Throwable}) \lor (T \equiv \text{any})): \\
(R'_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{et[l_a,l_b,T]} \\
(R'_n \Rightarrow E_{l_{i+1}}) \land \\
(R'_b \Rightarrow (f = \emptyset \Rightarrow (R'_b \Rightarrow E_{\text{break}})) \land \\
f \neq \emptyset \Rightarrow \forall i \in 1..k: \\
(\vdash \{ U^i \} s_i \{ V^i \} \land (V^i \Rightarrow U^{i+1} + 1) \land \\
(\forall T : \text{Type} : ((T \leq \text{Throwable}) \lor (T \equiv \text{any})): \\
(V^i_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{et[T]} ) \land \\
(R'_b \Rightarrow U^1_b) \land (V^k_b \Rightarrow E_{\text{break}}) \land \\
(\forall et_i = r^1_i + r^2_i + ... + r^m_i : (r^1_i \cap r^2_i \cap ... \cap r^m_i = \emptyset) \land (et_i \subseteq et))
\end{cases}
\right) \\
\Rightarrow \\
\forall l \in l_a \ldots l_g : \vdash \{ E_l \} I_l
\end{align*}}

Let \( f' \) be a list of Finally defined as:

\[
f' = \left[ Tree_2 \right]_{s_2} = \left[ \{ Q \} \right]_{s_2} = \{ R, R'_b, R'_e \}, \quad \text{getExcLines}(l_a, l_b, et) + f
\]

Let \( et' \) be an exception table defined as:

\[
et' = et + [l_a, l_b, l_d, \text{any}]
\]

By the induction hypothesis, we get:
\[ \vdash T_{S_1} \land \]
\[ [(I_{a_1} \ldots I_{l_{a-end}}), \ et_2] = \nabla_T (T_{S_1}, \ l_a, \ l_b, \ l_{break}, \ f', \ et') \land \]
\[ (\forall T : Type : ((T \leq \text{Throwable}) \lor (T \equiv \text{any})):\]
\[ (Q_e \land \text{exc}V \neq \text{null} \land s(0) = \text{exc}V) \Rightarrow E_{et'[l_a,l_b,T]} \land \]
\[ (Q_n \Rightarrow E_{lb}) \land \]
\[ (Q_b \Rightarrow (f' = \emptyset \Rightarrow (Q_b \Rightarrow E_{lbreat})) \land \]
\[ \left( f' \neq \emptyset \Rightarrow \forall i \in 1..k : \right) \]
\[ (\vdash \{U^i\} s_i \{V^i\} \land (V^i_k \Rightarrow U^i_k) \land \]
\[ (\forall T : Type : ((T \leq \text{Throwable}) \lor (T \equiv \text{any})):\]
\[ (V^i_e \land \text{exc}V \neq \text{null} \land s(0) = \text{exc}V) \Rightarrow E_{et''[T]} \land \]
\[ (Q_b \Rightarrow U^i_b) \land (V^i_b \Rightarrow E_{break}) \land \]
\[ (\forall et''_i = r^1_i + r^2_i + \ldots + r^m_i : (r^1_i \cap r^2_i \cap \ldots \cap r^m_i = \emptyset) \land (et''_i \subseteq et')) \]
\[ \Rightarrow \forall l \in l_a \ldots l_{a-end} : \vdash \{E_l\} I_l \]

We have \( \vdash T_{S_1} \). Since \( et' = et + [l_a, l_b, l_d, \text{any}] \), then we can prove:

\[ \forall T : Type : ((T \leq \text{Throwable}) \lor (T \equiv \text{any})):\]
\[ (Q_e \land \text{exc}V \neq \text{null} \land s(0) = \text{exc}V) \Rightarrow E_{et'[l_a,l_b,T]} \]

from the hypothesis (except for the type \text{any}). In the case of the type is \text{any}, we have to prove:

\[ \left( Q_e \land \tau(\text{exc}V) \leq T \land \right.
\[ \left. \text{exc}V \neq \text{null} \land s(0) = \text{exc}V \right) \Rightarrow E_{l_d} \]

since the target label of \( et'[l_a, l_b, T] \) is \( l_d \). By definition of \( E_{l_d} \), we have

\[ \left( Q_e \land \tau(\text{exc}V) \leq T \land \right.
\[ \left. \text{exc}V \neq \text{null} \land s(0) = \text{exc}V \right) \Rightarrow \left( \text{shift}(Q_e)[eTmp/\text{exc}V] \land \right.
\[ \left. \ldots \land eTmp = \text{exc}V \right) \]

The implication holds because \( Q_e \) does not refer to the stack, and then \( \text{shift}(Q_e) = Q_e \).

Finally, to be able to apply the first induction hypothesis, we need to prove:
\[(Q_b \Rightarrow (f' = \emptyset \Rightarrow (Q_b \Rightarrow E_{\text{break}}))) \land \\
(f' \neq \emptyset \Rightarrow \forall i \in 1..k : \\
(\vdash \{ U^i \} s_i \{ V^i \} \land (V^i_b \Rightarrow U^{i+1}_b) \land \\
(\forall T : \text{Type} : ((T \leq \text{Throwable}) \lor (T \equiv \text{any}))) : \\
(V^i_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{et''[T]} \land \\
(Q_b \Rightarrow U^1_b) \land (V^k_b \Rightarrow E_{\text{break}}) \land \\
(\forall et''_i = r^1_i + r^2_i + ... + r^m_i : (r^1_i \cap r^2_i \cap ... \cap r^m_i = \emptyset) \land (et''_i \subseteq et')))\]

Let \( et''_i \) be the exception table for the current finally block. The exception table for the last finally block is defined as \( et''_i = \text{getExcLines}(l_a, l_b, et) \). Using the definition of \( f' \), and the hypothesis, we need to prove:

\[
Q_b \Rightarrow U^0_b \quad \text{(D.18)} \\
V^k_b \Rightarrow E_{\text{break}} \quad \text{(D.19)} \\
V^0_b \Rightarrow U^1_b \quad \text{(D.20)} \\
(\forall T : \text{Type} : ((T \leq \text{Throwable}) \lor (T \equiv \text{any}))) : \\
(V^i_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{et''[T]} \quad \text{(D.21)} \\
(\forall et''_i = r^1_i + r^2_i + ... + r^m_i : (r^1_i \cap r^2_i \cap ... \cap r^m_i = \emptyset) \land (et''_i \subseteq et')) \quad \text{(D.22)}
\]

These expressions hold as follows:

[D.18]: \( Q_b \Rightarrow U^0_b \) holds since \( U^0_b \equiv Q_b \).

[D.19]: \( V^k_b \Rightarrow E_{\text{break}} \) holds because \( V^k_b \) is the same than the theorem hypothesis, and the implication holds in the theorem hypothesis.

[D.20]: \( V^0_b \Rightarrow U^1_b \): from theorem hypothesis, we know that \( R'_b \Rightarrow U^1_b \). The postcondition \( V^0_b \) is equal to \( R'_b \) then \( V^0_b \Rightarrow U^1_b \).

[D.21]: We have to prove

\[
(\forall T : \text{Type} : ((T \leq \text{Throwable}) \lor (T \equiv \text{any}))) : \\
(R^i_e \land \text{excV} \neq \text{null} \land s(0) = \text{excV}) \Rightarrow E_{et''[T]}
\]

because \( V^0_b \equiv R'_e \). We get the exceptions lines of \( et''_i \) from \( et \). Then the implication holds by hypothesis.

[D.22]: This property holds from the definition of \( \text{getExcLines} \) because \( \text{getExcLines} \) returns different exception lines.

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

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

Applying a similar reasoning to the translation of \( s_2 \), we get:
\( \forall l \in \{ l_a, \ldots, l_{b_{\text{end}}} \} : \vdash \{ E_l \} I_l \)

The instructions specifications at \( l_c, \ldots, l_g \) holds by definition of \( wp \). Joining the proofs, we get

\[ \forall l \in l_a \ldots l_g : \vdash \{ E_l \} I_l \]

\( \square \)

### D.2.6 Break Rule

The break rule is translated using Java Bytecode in Section 8.2.5 on page 167.

We have to prove:

\[ \vdash \{ P \} \text{break} \{ \text{false}, P, \text{false} \} \quad \land \quad ([I_{l_a} \ldots I_{l_b}), \ et_k] = \nabla_S \left( \{ P \} \text{break} \{ \text{false}, P, \text{false} \}, \ l_a, l_{b+1}, \ l_{\text{break}}, \ f, \ et'_0 \right) \quad \land \]

\( (\forall T : \text{Type} : ((T \leq \text{Throwable}) \lor (T \equiv \text{any})) : \)

\[ (\text{false} \land \text{exc} \neq \text{null} \land s(0) = \text{exc}V) \Rightarrow E_{et'_0[l_a,l_b,T]} \) \quad \land \]

\[ (\text{false} \Rightarrow E_{l_{b+1}}) \quad \land \]

\[ (P \Rightarrow (f = \emptyset \Rightarrow (P \Rightarrow E_{l_{\text{break}}})) \quad \land \]

\[ f \neq \emptyset \Rightarrow \forall i \in 1..k : \]

\[ \left( \begin{align*}
\vdash \{ U^i \} s_i \{ V^i \} & \quad \land \quad (V^i_0 \Rightarrow U^i_{l_{b+1}}) \quad \land \\
(\forall T : \text{Type} : ((T \leq \text{Throwable}) \lor (T \equiv \text{any})) : \\
(V^i \land \text{exc} \neq \text{null} \land s(0) = \text{exc}V) & \Rightarrow E_{et[T]} \) \quad \land \\
(P \Rightarrow U^i_{l_{b+1}}) & \quad \land \quad (V^i_0 \Rightarrow E_{\text{break}}) \quad \land \\
(\forall et_i = r^1 + r^2 + \ldots + r^m : (r^1 \cap r^2 \cap \ldots \cap r^m = \emptyset) \land (et_i \subseteq et'_0)) \end{align*} \right) \]

\[ \Rightarrow \quad \forall l \in l_a \ldots l_b : \vdash \{ E_l \} I_l \]

The translation of the break, first adds the translation of all finally blocks \( f_i \), and then it adds a \( \text{br} \) instruction. Let \( f_i = [T_{F_i}, \ et'_i] \) denote the \( i \)-th element of the list \( f \), where

\[ T_{F_i} = \frac{\text{Tree}_i}{\begin{array}{l} \{ U^i \} \\ s_i \{ V^i \} \end{array}} \]
D.2 Soundness of the Proof Translator using Java Bytecode

and $U_i^i$ and $V_i^i$ have the following form, which corresponds to the Hoare rule for try-finally:

$$
U_i^i = \left\{ (U_n^i \land \mathcal{X} \text{tmp} = \text{normal}) \lor (U_b^i \land \mathcal{X} \text{tmp} = \text{break}) \lor \right\} \\
\left\{ (U_b^i[\text{e tmp} / \text{exc} V] \land \mathcal{X} \text{tmp} = \text{exc} \land \text{e tmp} = \text{exc} \time V ) \right\}
$$

$$
V_i^i = \left\{ (V_n^i \land \mathcal{X} \text{tmp} = \text{normal}) \lor (V_b^i \land \mathcal{X} \text{tmp} = \text{break}) \lor (V_e^i \land \mathcal{X} \text{tmp} = \text{exc}) \right\}, \ V_b^i, \ V_e^i
$$

We can prove that the translation of $f_i$ by induction hypothesis. We show that the hypotheses holds for $f_i$ arbitrary:

$$[b_i, e_t^i] = \nabla_S (T_{F_i}, \ l_{a_i}, l_{a_i+1}, \ l_{b_i}, \ f_{i+1} + ... + f_k, \ \text{divide}(e_t^{i-1}, \ e_t^i, \ l_{a_i}, l_{a_i+1}))$$

Let $et = \text{divide}(e_t^{i-1}, \ e_t^i, \ l_{a_i}, l_{a_i+1})$. We have to prove:

$$\vdash T_{F_i} \land
\left[(I_{b_i}...I_{b_{\text{end}}}), \ e_{t_k}\right] = \nabla_S (T_{F_i}, \ l_{f_i}, l_{b_{e}}, \ l_{b_{\text{break}}}f_{i+1}...f_k, \ et) \land
(\forall T : \text{Type} : ((T \leq \text{Throwable}) \lor (T \equiv \text{any})):
\left(V_i^i \land \text{exc} V \neq \text{null} \land s(0) = \text{exc} V \Rightarrow E_{et[I_{b_i}...I_{b_{\text{end}}}, T]} \right) \land
\left(V_b^i \Rightarrow E_{l_{b_{e}}} \right) \land
(V_b^i \Rightarrow (f = \emptyset \Rightarrow (V_b^i \Rightarrow E_{b_{\text{break}}})) \land
\left(f \neq \emptyset \Rightarrow \forall i \in 1..k : \left(\vdash \left\{ U_i^i \right\} s_i V_i^i \land (V_b^i \Rightarrow U_b^{i+1}) \land
(\forall T : \text{Type} : ((T \leq \text{Throwable}) \lor (T \equiv \text{any})):
\left(V_i^i \land \text{exc} V \neq \text{null} \land s(0) = \text{exc} V \Rightarrow E_{et_i[T]} \right) \land
(V_b^i \Rightarrow U_b^{i+1}) \land
(V_b^i \Rightarrow E_{b_{\text{break}}}) \land
(\forall et_i = r_{i_1} \land r_{i_2} \land ... \land r_{i_m} : (r_{i_1} \land r_{i_2} \land ... \land r_{i_m} = \emptyset) \land (et_i \subseteq et)) \right) \Rightarrow
\forall l \in l_{f_i}...l_{f_{\text{end}}} : \vdash \left\{ E_i \right\} I_l
$$

We can prove:

$$\forall T : \text{Type} : ((T \leq \text{Throwable}) \lor (T \equiv \text{any})):
\left(V_i^i \land \text{exc} V \neq \text{null} \land s(0) = \text{exc} V \Rightarrow E_{et[I_{b_i}...I_{b_{\text{end}}}, T]} \right)$$
using the hypothesis (case \( f \neq \emptyset \)):

\[
(\forall T : Type : ((T \preceq \text{Throwable}) \lor (T \equiv \text{any})) : \\
(V^i_c \land \text{exc}V \neq \text{null} \land s(0) = \text{exc}V) \Rightarrow E_{\text{et}(T)})
\]

and Lemma \ref{lem:et} that says \( et[l_{fi}, l_{fi\_end}, T] = et[T] \).

Showing that \( V^i_{in} \Rightarrow E_{inext} \) holds is equivalent to prove \( V^i_{in} \Rightarrow E_{l_{fi+1}} \). The precondition \( E_{l_{fi+1}} \) is defined as \( U^i_{i+1} \). Then, we have to prove \( V^i_{in} \Rightarrow U^i_{1+b} \) and \( V^i_{k*b} \Rightarrow E_{l_{break}} \) and \( \forall et_i = r^i_1 + r^i_2 + \ldots + r^i_m : (r^i_1 \cap r^i_2 \cap \ldots \cap r^i_m = \emptyset) \land (et_i \subseteq et) \)

by hypothesis. The translation changes the list \( f \) by \( f_i \ldots f_k \). But this property holds for \( f \ldots f_k \).

Now we can apply the induction hypothesis and get \( \forall l \in l_{fi} \ldots l_{fi\_end} : \vdash \{E_l\} I_l \)

Finally, we have to prove that:

\[
P \Rightarrow U^i_{b} \text{ and for all } i : V^i_{b} \Rightarrow U^i_{b+1} \text{ and } V^k_{b} \Rightarrow E_{l_{break}}
\]

These implications hold by hypothesis.

\( \square \)
Bibliography


[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison Wesley, 1994.


CURRICULUM VITAE

General Information

Name: Darío Martín Nordio
Date of birth: March 30th, 1979
Nationality: Argentinean / Italian
E-mail address: martin.nordio@inf.ethz.ch
Web page: http://se.ethz.ch/people/nordio/index.html

Education

- Master in Computer Science
  Universidad de la República, Uruguay (2005)
- Licentiate in Computer Science
  National University of Río Cuarto, Argentina (2002)
- Computer Analyst
  National University of Río Cuarto, Argentina (2001)

Career History

- July 2005 - November 2009: PhD student at the Chair of Software Engineering, ETH Zurich (Swiss Federal Institute of Technology Zurich). Switzerland.
- August 2002 - June 2005: Teaching assistant at National University of Río Cuarto. Argentina.