Proofs and Proof Transformations for Object-Oriented Programs

Martin Nordio

2009

Diss. ETH N^0 18689

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[1], I started to prepare my backpack. While waiting for my boat to depart, *Jorge Aguirre* suggested that I start a shorter trip[2] 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 Üetliberg 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[3]: *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*,

[1] *Licenciate* in computer science

[2] Master in computer science

[3] Past and present members of the chair of Software Engineering and Chair of Programming Methodology at ETH Zurich

Manuel Oriol, *Michela Pedroni*, *Marco Piccioni*, *Max (Yu) Pei*, *Nadia Polikarpova*, *Arsenii Rudich*, *Joseph Ruskiewicz*, *Bernd Schoeller*, *Marco Trudel*, *Julian Tschannen*, *Stephan van Staden*, *Jason (Yi) Wei*, *Scott West*, *Volkan Arslan*, *Till Bay*, and *Patrick Eugster*. 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

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 *"jitting"*. 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-transfoming 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 puó produrre risultati incorretti, che potrebbero compromettere la sicurezza e la correttezza dei sistemi di software. Un aspetto importante da considerare e' 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 proprieta' di tipo, come nel PCC originale, ma e' in grado di esprimere correttezza funzionale.

Questa tesi sviluppa gli aspetti fondamentali di *proof-transfoming 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.

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) [85] 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.

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

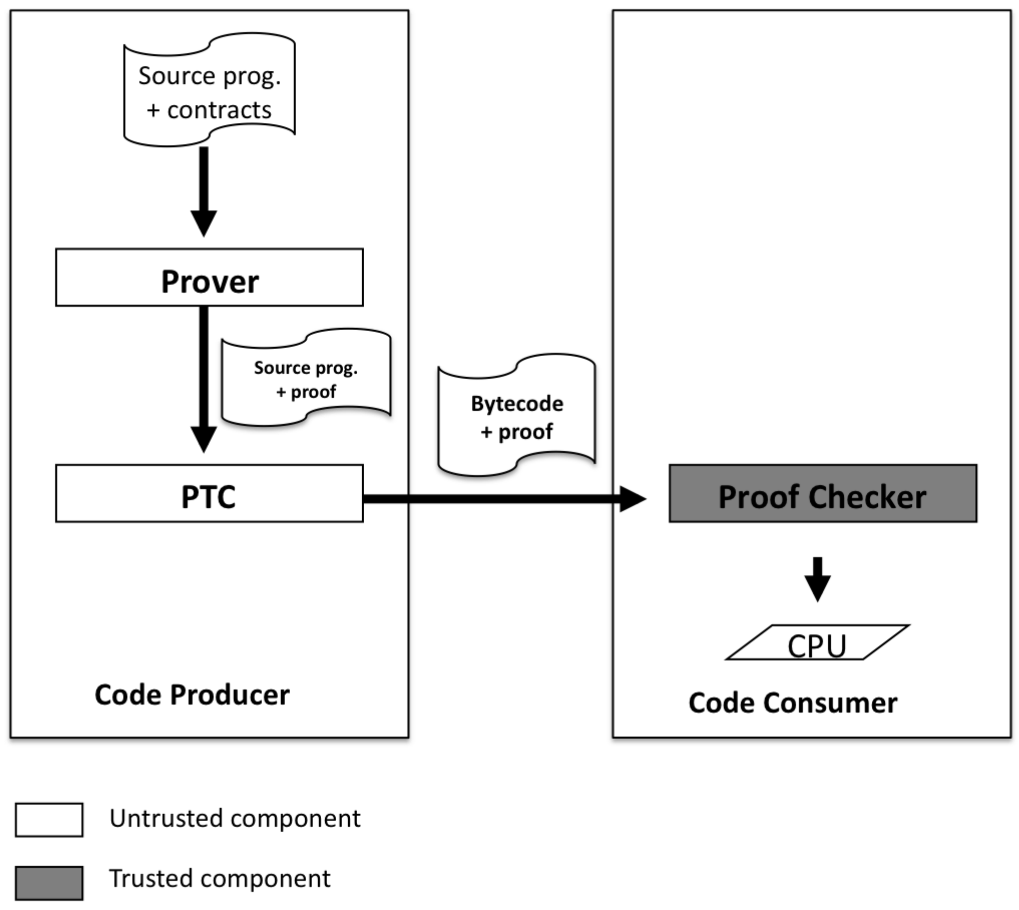

Fig. 2.1: General Architecture of Proof-Transforming Compilation.

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.

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 `try-catch`, `try-finally`, and `break` instructions. This subset is interesting from the semantics point of view due to the combination of `while`, `break`, `try-catch`, and `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 `retry` instructions in a `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[1].

Function Objects. The second main contribution is a methodology to reason about *function objects* such as *delegates* in C#, *agents* in Eiffel, and *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 [63].

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 byte-

[1]Java Hoare triples also have an extra postcondition for `break` instructions.

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].

Part II

Proofs

Chapter 3

A Sound and Complete Logic for Object-Oriented Programs

Program verification relies on a formal semantics of the programming language, typically a program logic such as Hoare logic [48]. 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 [108]. 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 [52, 57, 109] 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 [108]. 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 4. The third and final part of the logic is presented in Section 3.3. 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[1], 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[2]. 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*,

[1]Mate is pronounced |*'mate*|; the name comes from the Argentinean mate: `http://en.wikipedia.org/wiki/Mate_(beverage)`

[2]The necessary checks are delegated to the compiler.

Program	::=	*ClassDecl∗*
ClassDecl	::=	`class` *ClassId* [`inherit` *Type*] *ClassBody* `end`
Type	::=	*BoolT* | *IntT* | *ClassId* | *VoidT*
ClassBody	::=	*MemberDecl∗*
MemberDecl	::=	*AttributeId* *Type*
	|	*Routine*
Routine	::=	*RoutineId* (*Type*) : *Type*
		[`local` *list_of* (*VarId* : *Type*)]
		`do`
		Instr
		`end`
Instr	::=	*VarId* := *ExpE*
	|	*Instr*; *Instr*
	|	`until` *BoolExp* `loop` *Instr* `end`
	|	`if` *BoolExp* `then` *Instr* `else` *Instr* `end`
	|	`check` *BoolExp* `end`
	|	*VarId* := `create` {*Type*}.*make* (*Exp*)
	|	*VarId* := *VarId*.*Type*@*AttributeId*
	|	*VarId*.*Type*@*AttributeId* := *Exp*
	|	*VarId* := *VarId*.*Type* : *RoutineId* (*Exp*)
Exp	::=	*Literal* | *VarId* | *Current* | *Result* | *Exp* *Op* *Exp* | *BoolExp*
BoolExp	::=	*Literal* | *VarId* | *Current* | *Result* | *BoolExp* *Bop* *BoolExp*
		| *Exp* *CompOp* *Exp*
Op	::=	$+$ | $-$ | $*$
Bop	::=	`and` | `or`
CompOp	::=	$<$ | $>$ | $<=$ | $>=$ | $=$ | $/=$
ExpE	::=	*Exp* | *ExpE* //*ExpE*

Fig. 3.1: Syntax of the Mate Language.

the instruction s is executed (opposite to the `do` s `while` e instruction[3], 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:

[3]The `do` s `while` e instruction is also known as `repeat` s `until` e instruction (for example in Pascal).

$$\begin{array}{lll}\textbf{data type } \mathit{Value} = & \textbf{boolV } \mathit{Bool} \\ & \mid \textbf{intV } \mathit{Int} \\ & \mid \textbf{objV } \mathit{ClassId\ ObjId} \\ & \mid \textbf{voidV}\end{array}$$

The function $\tau : \mathit{Value} \rightarrow \mathit{Type}$ returns the dynamic type of a value. This function is defined as follows:

$$\begin{array}{ll}\tau : \mathit{Value} \rightarrow \mathit{Type} & \\ \quad \tau(\textbf{boolV } b) & = \mathit{BoolT} \\ \quad \tau(\textbf{intV } n) & = \mathit{IntT} \\ \quad \tau(\textbf{objV } \mathit{cId\ oId}) & = \mathit{cId} \\ \quad \tau(\textbf{voidV}) & = \mathit{VoidT}\end{array}$$

The function *init* yields a default value for every type. The default value of boolean is *false*, the default value of integer is zero, and the default value of references is *void*. Its definition is as follows:

$$\begin{array}{ll}\mathit{init} : \mathit{Type} \rightarrow \mathit{Value} & \\ \quad \mathit{init}(\mathit{BoolT}) & = (\textbf{boolV } \mathit{false}) \\ \quad \mathit{init}(\mathit{IntT}) & = (\textbf{intV } 0) \\ \quad \mathit{init}(\mathit{cId}) & = \textbf{voidV} \\ \quad \mathit{init}(\mathit{VoidT}) & = \textbf{voidV}\end{array}$$

The state of an object is defined by the values of its attributes. The sort *Attribute* defines the attribute declaration $T@a$ where a is an attribute declaration in the class T. We use a sort *Location*, and a function *instvar* where $\mathit{instvar}(V, T@a)$ 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 undef. The datatype definitions and the signature of *instvar* are the following:

$$\begin{array}{ll}\textbf{data type } \mathit{Attribute} & = \textbf{Attr } \mathit{Type\ AttributeId} \\ \textbf{data type } \mathit{Location} & = \textbf{Loc } \mathit{ObjId\ AttributeId}\end{array}$$

$$\mathit{instvar} : \mathit{Value} \times \mathit{Attribute} \rightarrow \mathit{Location} \cup \{\mathit{undef}\}$$

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 *ObjectStore*. We use the object store presented by Poetzsch-Heffter and Müller [107, 108]. The following operations apply to the object store: $OS(L)$ denotes reading the location L

in the object store OS; $alive(O, OS)$ yields *true* if and only if the object O is allocated in the object store OS; $new(OS, C)$ 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; $OS\langle C\rangle$ denotes the store after allocating a new object of type C.

$$\begin{array}{rcl}
\_(\_) & : & ObjectStore \times Location \rightarrow Value \\
alive & : & Value \times ObjectStore \rightarrow Bool \\
new & : & ObjectStore \times ClassId \rightarrow Value \\
\_\langle\_ := \_\rangle & : & ObjectStore \times Location \times Value \rightarrow ObjectStore \\
\_\langle\_\rangle & : & ObjectStore \times ClassId \rightarrow ObjectStore
\end{array}$$

In the following, we present Poetzsch-Heffter and Müller's axiomatization [107] of these functions with a brief description. The function $obj : Location \rightarrow Value$ takes a location and yields the object it belongs to. The function $ltyp : Location \rightarrow Type$ yields the dynamic type of a location. The function $ref : Value \rightarrow Bool$ 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:*
$\forall OS \in ObjectStore,\ L_1, L_2 \in Location,\ X \in Value :$
$L_1 \neq L_2 \ \Rightarrow OS\langle L_1 := X\rangle(L_2) = OS(L_2)$

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

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

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

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

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

Axiom 7. *An object is alive if only if the object was alive before an allocation or the object is the new object:*
$\forall\ OS \in ObjectStore,\ X \in Value,\ cId \in ClassId :$
$alive(X, OS\langle cId\rangle) \Leftrightarrow alive(X, OS)\ \vee\ X = new(OS, cId)$

Axiom 8. *Objects held by locations are alive:*
$\forall\ OS \in \mathit{ObjectStore},\ L \in \mathit{Location}:\ \mathit{alive}(OS(L), OS)$

Axiom 9. *Non reference values are alive:*
$\forall\ OS \in \mathit{ObjectStore}:\ \neg \mathit{ref}(X) \Rightarrow \mathit{alive}(X, OS)$

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

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

3.1.3 Operational Semantics

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

$$\begin{array}{lcl} \mathit{State} & \equiv & \mathit{Local}\ \times\ \mathit{Heap} \\ \mathit{Local} & \equiv & \mathit{VarId} \cup \{\mathit{Current}, p, \mathit{Result}\} \rightarrow \mathit{Value} \cup \{\mathit{undef}\} \\ \mathit{Heap} & \equiv & \{\$\} \rightarrow \mathit{ObjectStore} \end{array}$$

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

For $\sigma \in \mathit{State}$, $\sigma(e)$ denotes the evaluation of the expression e in the state σ. Its signature is the following:

$$\sigma : \mathit{ExpE} \rightarrow \mathit{Value} \cup \{\mathit{exc}\}$$

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

The transitions of the operational semantics have the form:

$$\langle \sigma, S \rangle \rightarrow \sigma', \chi$$

where σ and σ' are states, S is an instruction, and χ is the current status of the program. The value of χ can be either the constant *normal* or *exc*. The variable χ is required to treat abrupt termination. The transition $\langle \sigma, S \rangle \rightarrow \sigma', \mathit{normal}$ expresses that executing the instruction S in the state σ terminates normally in the state σ'. The transition $\langle \sigma, S \rangle \rightarrow \sigma', \mathit{exc}$ expresses that executing the instruction S in the state σ terminates with an exception in the state σ'. In the following, we present the operational 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 σ.

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 σ' 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

$$\frac{\sigma(e) = exc}{\langle \sigma, x := e \rangle \rightarrow \sigma, exc} (3.2.1) \qquad \frac{\sigma(e) \neq exc}{\langle \sigma, x := e \rangle \rightarrow \sigma[x := \sigma(e)], normal} (3.2.2)$$

Compound

$$\frac{\langle \sigma, s_1 \rangle \rightarrow \sigma', exc}{\langle \sigma, s_1; s_2 \rangle \rightarrow \sigma', exc} (3.2.3) \qquad \frac{\langle \sigma, s_1 \rangle \rightarrow \sigma', normal \quad \langle \sigma', s_2 \rangle \rightarrow \sigma'', \chi}{\langle \sigma, s_1; s_2 \rangle \rightarrow \sigma'', \chi} (3.2.4)$$

Check Instruction

$$\frac{\sigma(e) = true}{\langle \sigma, \texttt{check}\ e\ \texttt{end} \rangle \rightarrow \sigma, normal} (3.2.5) \qquad \frac{\sigma(e) = false}{\langle \sigma, \texttt{check}\ e\ \texttt{end} \rangle \rightarrow \sigma, exc} (3.2.6)$$

Conditional Instruction

$$\frac{\sigma(e) = true \quad \langle \sigma, s_1 \rangle \rightarrow \sigma', \chi}{\langle \sigma, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end} \rangle \rightarrow \sigma', \chi} (3.2.7)$$

$$\frac{\sigma(e) = false \quad \langle \sigma, s_2 \rangle \rightarrow \sigma', \chi}{\langle \sigma, \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end} \rangle \rightarrow \sigma', \chi} (3.2.8)$$

Loop Instruction

$$\frac{\sigma(e) = true}{\langle \sigma, \texttt{until}\ e\ \texttt{loop}\ s_1\ \texttt{end} \rangle \rightarrow \sigma, normal} (3.2.9)$$

$$\frac{\sigma(e) = false \quad \langle \sigma, s_1 \rangle \rightarrow \sigma', exc}{\langle \sigma, \texttt{until}\ e\ \texttt{loop}\ s_1\ \texttt{end} \rangle \rightarrow \sigma', exc} (3.2.10)$$

$$\frac{\begin{array}{c}\sigma(e) = false \quad \langle \sigma, s_1 \rangle \rightarrow \sigma', normal \\ \langle \sigma', \texttt{until}\ e\ \texttt{loop}\ s_1\ \texttt{end} \rangle \rightarrow \sigma'', \chi\end{array}}{\langle \sigma, \texttt{until}\ e\ \texttt{loop}\ s_1\ \texttt{end} \rangle \rightarrow \sigma'', \chi} (3.2.11)$$

Read Attribute Instruction

$$\frac{\sigma(y) \neq voidV}{\langle \sigma, x := y.T@a \rangle \rightarrow \sigma[x := \sigma(\$)\ (instvar(\sigma(y), T@a))], normal} (3.2.12)$$

$$\frac{\sigma(y) = voidV}{\langle \sigma, x := y.T@a \rangle \rightarrow \sigma, exc} (3.2.13)$$

Write Attribute Instruction

$$\frac{\sigma(y) \neq voidV}{\langle \sigma, y.T@a := e \rangle \rightarrow \sigma[\$:= \sigma(\$)\langle instvar(\sigma(y), T@a) := \sigma(e) \rangle], normal} (3.2.14)$$

$$\frac{\sigma(y) = voidV}{\langle \sigma, y.T@a := e \rangle \rightarrow \sigma, exc} (3.2.15)$$

Fig. 3.2: Operational Semantics for Basic Instructions

handling.

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:

$$\begin{aligned} &\mathit{impl} : \mathit{Type} \times \mathit{RoutineId} \rightarrow \mathit{RoutineImpl} \cup \{\mathit{undef}\} \\ &\mathit{body} : \mathit{RoutineImpl} \rightarrow \mathit{Instr} \end{aligned}$$

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

$$\frac{\sigma(y) = \mathit{voidV}}{\langle \sigma, x := y.T{:}m(e) \rangle \rightarrow \sigma, \mathit{exc}} \tag{3.1}$$

$$\frac{\sigma(y) \neq \mathit{voidV} \qquad \langle \mathit{initSt}[\mathit{Current} := \sigma(y), p := \sigma(e), \$:= \sigma(\$)], \mathit{body}(\mathit{impl}(\tau(\sigma(y)), m)) \rangle \rightarrow \sigma', \mathit{normal}}{\langle \sigma, x := y.T{:}m(e) \rangle \rightarrow \sigma[x := \sigma'(\mathit{Result}), \$:= \sigma'(\$)], \mathit{normal}} \tag{3.2}$$

$$\frac{\sigma(y) \neq \mathit{voidV} \qquad \langle \mathit{initS}[\mathit{Current} := \sigma(y), p := \sigma(e), \$:= \sigma(\$)], \mathit{body}(\mathit{impl}(\tau(\sigma(y)), m)) \rangle \rightarrow \sigma', \mathit{exc}}{\langle \sigma, x := y.T{:}m(e) \rangle \rightarrow \sigma[\$:= \sigma'(\$)], \mathit{exc}} \tag{3.3}$$

In rule (3.1), since the target y is *Void*, the state σ 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 τ. 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 σ' (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):

```
rId (x : T) : T'
  local
    v_1 : T_1; ... v_i : T_i
  do
    s
  end
```

the function *body* returns the following result:

$$\texttt{local}\ \ v_1 : T_1;\ ...\ v_i : T_i; \mathit{Result} : T';\ \ s$$

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

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

$$\frac{\langle \sigma[v_1 := \mathit{init}(T_1), ..., v_n := \mathit{init}(T_n)],\ \ s\rangle \rightarrow \sigma', \chi}{\langle \sigma, \texttt{local}\ \ v_1 : T_1;\ ...\ v_n : T_n;\ \ s\rangle \rightarrow \sigma', \chi} \tag{3.4}$$

Creation Instruction

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

$$\frac{\langle \mathit{initSt}[\mathit{Current} := \mathit{new}(\sigma(\$), T), \$:= \sigma(\$)\langle T\rangle, p := \sigma(e)],\ \ \mathit{body}(\mathit{impl}(T, \mathit{make}))\rangle \rightarrow \sigma', \mathit{normal}}{\langle \sigma, x := \texttt{create}\ \ T.\mathit{make}(e)\rangle \rightarrow \sigma[x := \sigma'(\mathit{Current}), \$:= \sigma(\$)], \mathit{normal}} \tag{3.5}$$

$$\frac{\langle \mathit{initSt}[\mathit{Current} := \mathit{new}(\sigma(\$), T), \$:= \sigma(\$)\langle T\rangle, p := \sigma(e)],\ \ \mathit{body}(\mathit{impl}(T, \mathit{make}))\rangle \rightarrow \sigma', \mathit{exc}}{\langle \sigma, x := \texttt{create}\ \ T.\mathit{make}(e)\rangle \rightarrow \sigma[\$:= \sigma(\$)], \mathit{exc}} \tag{3.6}$$

In rule (3.5), the routine *make* terminates normally, and the object x is updated with the *Current* object in σ'. 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 *new* which creates the new object initializing its attributes. If the routine *make* triggers an exception (rule 3.6), the state produced by the creation instruction is σ', and x is not updated.

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 χ 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 byteocde 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 *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:

$$embed(e)(\sigma)\ \Leftrightarrow\ \sigma(e) = \mathbf{boolV}\ true$$

This embedding is trivial, thus it is omitted.

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

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

$$\begin{array}{ll} \mathit{safe} : \mathit{ExpE} \rightarrow \mathit{Exp} & \\ \mathit{safe}\ (e_1\ \mathit{oper}\ e_2) = \mathit{safe}(e_1)\ \textbf{and}\ \mathit{safe}(e_2)\ \ \textbf{and}\ \mathit{safe\_op}\ (\mathit{oper}, e_1, e_2) & \\ \mathit{safe}\ (e) \qquad\qquad = \mathit{true} & \end{array}$$

$$\begin{array}{l} \mathit{safe\_op} : \mathit{Op} \times \mathit{ExpE} \times \mathit{ExpE} \rightarrow \mathit{Exp} \\ \mathit{safe\_op}\ (\ //,\ \ e_1,\ e_2) = (e_2\ / = 0) \\ \mathit{safe\_op}\ (\mathit{oper},\ e_1,\ e_2) = \mathit{true} \end{array}$$

Lemma 1. *For each expression e, $safe(e)$ satisfies:*
$\sigma(\mathit{safe}(e)) = \mathit{true}\ \Leftrightarrow \sigma(e) \neq \mathit{exc}$

Proof. By induction on the structure of e, and definition of *safe*.

□

Lemma 2 (Substitution)**.** *If the expression e is a **safe expression**, then:*

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

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

□

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 Σ that represents the constant symbols of these entities. Given a Mate program, Σ 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 *Value* and *ObjectStore*. Preconditions and postconditions of Hoare triples are formulas over $\Sigma\ \cup\ \{\mathit{Current}, p, \mathit{Result}\}\ \cup\ \mathit{Var}(r)$ where r is a routine, and $\mathit{Var}(r)$ denotes the set of local variables of r. Note that we assume $\mathit{Var}(r)$ does not include the *Result* variable, it only includes the local variables declared by the programmer. Routine preconditions are formulas over $\Sigma\ \cup\ \{\mathit{Current}, p, \$\}$, routine postconditions for normal termination are formulas over $\Sigma\ \cup\ \{\mathit{Result}, \$\}$, 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 *sequents* of the form $\mathcal{A} \vdash \mathbf{A}$ where $\mathcal{A}$ is a set of routine annotations and $\mathbf{A}$ is a triple. Triples in $\mathcal{A}$ are called assumptions of the sequent, and $\mathbf{A}$ 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.

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 $P \ \wedge \ e$ holds. If e is *false*, an exception is triggered and $P \ \wedge \ \neg e$ 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:

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

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:

$$\dfrac{}{\vdash \left\{ \begin{array}{l} (safe(e) \ \wedge \ P[e/x]) \vee \\ (\neg safe(e) \ \wedge \ Q_e) \end{array} \right\} \ x \ := \ e \ \ \{ \ P \ , \ Q_e \ \}}$$

Compound Rule:

$$\dfrac{\mathcal{A} \vdash \{ \ P \ \} \ \ s_1 \ \ \{ \ Q_n \ , \ R_e \ \} \qquad \mathcal{A} \vdash \{ \ Q_n \ \} \ \ s_2 \ \ \{ \ R_n \ , \ R_e \ \}}{\mathcal{A} \vdash \{ \ P \ \} \ \ s_1; s_2 \ \ \{ \ R_n \ , \ R_e \ \}}$$

Conditional Rule:

$$\dfrac{\mathcal{A} \vdash \{ \ P \ \wedge \ e \ \} \ \ s_1 \ \ \{ \ Q_n \ , \ Q_e \ \} \qquad \mathcal{A} \vdash \{ \ P \ \wedge \ \neg e \ \} \ \ s_2 \ \ \{ \ Q_n \ , \ Q_e \ \}}{\mathcal{A} \vdash \{ \ P \ \} \ \ \texttt{if} \ e \ \texttt{then} \ s_1 \ \texttt{else} \ s_2 \ \texttt{end} \ \ \{ \ Q_n \ , \ Q_e \ \}}$$

Check Axiom:

$$\dfrac{}{\mathcal{A} \vdash \{ \ P \ \} \ \ \texttt{check} \ e \ \texttt{end} \ \ \{ \ (P \ \wedge \ e \ ) \ , \ (P \ \wedge \ \neg e \ ) \ \}}$$

Loop Rule:

$$\dfrac{\mathcal{A} \vdash \{ \ \neg e \ \wedge \ I \ \} \ \ s_1 \ \ \{ \ I \ , \ R_e \ \}}{\mathcal{A} \vdash \{ \ I \ \} \ \ \texttt{until} \ e \ \texttt{loop} \ s_1 \ \texttt{end} \ \ \{ \ (I \ \wedge \ e) \ , \ R_e \ \}}$$

Read Attribute Axiom:

$$\dfrac{}{\mathcal{A} \vdash \left\{ \begin{array}{l} (y \neq Void \ \wedge \ P[\$(instvar(y, T@a))/x]) \vee \\ (y = Void \ \wedge \ Q_e) \end{array} \right\} \ x := y.T@a \ \ \{ \ P \ , \ Q_e \ \}}$$

Write Attribute Axiom:

$$\dfrac{}{\mathcal{A} \vdash \left\{ \begin{array}{l} (y \neq Void \ \wedge \ P[\$ \ \langle instvar(y, T@a) := e \ \rangle/\$]) \vee \\ (y = Void \ \wedge \ Q_e) \end{array} \right\} \ y.T@a := e \ \ \{ \ P \ , \ Q_e \ \}}$$

Fig. 3.3: Axiomatic Semantics for Basic Instructions

Invoc-var-rule:

$$\dfrac{\mathcal{A} \vdash \{ \ P \ \} \ \ x := y.T{:}m(e) \ \ \{ \ Q_n \ , \ Q_e \ \}}{\mathcal{A} \vdash \{ \ P[w/Z] \ \} \ \ x := y.T{:}m(e) \ \ \{ \ Q_n[w/Z] \ , \ Q_e[w/Z] \ \}}$$

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

Class Rule:

$$\frac{\begin{array}{c}\mathcal{A} \vdash \left\{ \ \tau(\mathit{Current}) = T \ \wedge \ P \ \right\} \ \ \mathit{impl}(T, m) \ \ \left\{ \ Q_n \ , \ Q_e \ \right\} \\ \mathcal{A} \vdash \left\{ \ \tau(\mathit{Current}) \prec T \ \wedge \ P \ \right\} \ \ T{:}m \ \ \left\{ \ Q_n \ , \ Q_e \ \right\}\end{array}}{\mathcal{A} \vdash \left\{ \ \tau(\mathit{Current}) \preceq T \ \wedge \ P \ \right\} \ \ T{:}m \ \ \left\{ \ Q_n \ , \ Q_e \ \right\}}$$

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 $S \prec T \ \Leftrightarrow_{\mathit{def}} \ S \preceq T \wedge S \neq T$.

Subtype Rule:

$$\frac{\begin{array}{c} S \preceq T \\ \mathcal{A} \vdash \left\{ \ P \ \right\} \ \ S{:}m \ \ \left\{ \ Q_n \ , \ Q_e \ \right\}\end{array}}{\mathcal{A} \vdash \left\{ \ \tau(\mathit{Current}) \preceq S \ \wedge \ P \ \right\} \ \ T{:}m \ \ \left\{ \ Q_n \ , \ Q_e \ \right\}}$$

Routine Implementation

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

Routine Implementation Rule:

$$\frac{\mathcal{A}, \{P\} \ \ T@m \ \ \{Q_n, \ Q_e\} \vdash \ \{ \ P \ \} \ \ \mathit{body}(T@m) \ \ \{ \ Q_n \ , \ Q_e \ \}}{\mathcal{A} \vdash \{ \ P \ \} \ \ T@m \ \ \{ \ Q_n \ , \ Q_e \ \}}$$

Local

In Mate, local variables have default values. To initialize local variables we use the function *init*. The following rule is used to initializes default values:

Local Rule:

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

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 *make*. In the following rule, the object creation is expressed by the replacement $new(\$, T)/Current, \$\langle T \rangle/\$$. The replacement e/p is added due to the routine invocation. Finally, in the postcondition, the *Current* object is replaced by x . The rule is the following:

Creation Rule:

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

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 *false axiom* allows us to prove anything assuming *false*. The *assumpt-axiom* allows proving a triple **A** assuming the triple holds. The *assumpt-intro-axiom* and the *assumpt-elim-axiom* allow introducing and eliminating a triple $\mathbf{A}_0$ in the hypothesis.

The *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 *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 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 *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.

3.1.5 Example

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

Assumpt-axiom

$$\frac{}{\mathbf{A} \vdash \mathbf{A}}$$

False axiom

$$\frac{}{\vdash \{\ \mathit{false}\ \}\ \ s_1\ \ \{\ \mathit{false}\ ,\ \mathit{false}\ \}}$$

Assumpt-intro-axiom

$$\frac{\mathcal{A} \vdash \mathbf{A}}{\mathbf{A_0}, \mathcal{A} \vdash \mathbf{A}}$$

Assumpt-elim-axiom

$$\frac{\begin{array}{c}\mathcal{A} \vdash \mathbf{A_0} \\ \mathbf{A_0}, \mathcal{A} \vdash \mathbf{A}\end{array}}{\mathcal{A} \vdash \mathbf{A}}$$

Strength

$$\frac{\begin{array}{c}P' \Rightarrow P \\ \mathcal{A} \vdash \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}\end{array}}{\mathcal{A} \vdash \{\ P'\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}}$$

Weak

$$\frac{\begin{array}{c}\mathcal{A} \vdash \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \} \\ Q_n \Rightarrow Q'_n \\ Q_e \Rightarrow Q'_e\end{array}}{\mathcal{A} \vdash \{\ P\ \}\ \ s_1\ \ \{\ Q'_n\ ,\ Q'_e\ \}}$$

Conjunction

$$\frac{\begin{array}{c}\mathcal{A} \vdash \{\ P'\ \}\ \ s_1\ \ \{\ Q'_n\ ,\ Q'_e\ \} \\ \mathcal{A} \vdash \{\ P''\ \}\ \ s_1\ \ \{\ Q''_n\ ,\ Q''_e\ \}\end{array}}{\mathcal{A} \vdash \{\ P' \wedge P''\ \}\ \ s_1\ \ \{\ Q'_n \wedge Q''_n\ ,\ Q'_e \wedge Q''_e\ \}}$$

Disjunction

$$\frac{\begin{array}{c}\mathcal{A} \vdash \{\ P'\ \}\ \ s_1\ \ \{\ Q'_n\ ,\ Q'_e\ \} \\ \mathcal{A} \vdash \{\ P''\ \}\ \ s_1\ \ \{\ Q''_n\ ,\ Q''_e\ \}\end{array}}{\mathcal{A} \vdash \{\ P' \vee P''\ \}\ \ s_1\ \ \{\ Q'_n \vee Q''_n\ ,\ Q'_e \vee Q''_e\ \}}$$

Invariant

$$\frac{\mathcal{A} \vdash \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}}{\mathcal{A} \vdash \{\ P \wedge W\ \}\ \ s_1\ \ \{\ Q_n \wedge W\ ,\ Q_e \wedge W\ \}}$$

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

Substitution

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

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

all-rule

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

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

ex-rule

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

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 ≥ 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 *sum* satisfies the following specifications:

$$\left\{\ n > 1\ \right\}\quad \textit{MATH:sum}\quad \left\{\ \textit{Result} = (n * (n+1))/2\ ,\ \textit{false}\ \right\}$$

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

$$\left\{\ n > 1\ \right\}\quad \textit{body(MATH@sum)}\quad \left\{\ \textit{Result} = (n * (n+1))/2\ ,\ \textit{false}\ \right\}$$

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 *MATH@sum* satisfies its specification:

$$\frac{\left\{\ \tau(\textit{Current}) = \textit{MATH}\ \wedge\ n > 1\ \right\}\ \ \textit{body(MATH@sum)}\ \ \left\{\ \textit{Result} = (n*(n+1))/2\ ,\ \textit{false}\ \right\}}{\left\{\ \tau(\textit{Current}) = \textit{MATH}\ \wedge\ n > 1\ \right\}\ \ \textit{MATH@sum}\ \ \left\{\ \textit{Result} = (n*(n+1))/2\ ,\ \textit{false}\ \right\}}$$

The class *MATH* does not have any descendent. Thus, we can show:

```
$sum\ (n\text{: } INTEGER)\text{: } INTEGER$
    **require**
        $positive : n>1$
    **local**
        $i\text{: } INTEGER$
    **do**
        $\{ n > 1 \}$
        **from**
            $\{ n > 1 \}$
            **Result** $:= 1$
            $\{ n > 1$ **and Result** $= 1 \}$
            $i := 2$
            $\{ n > 1$ **and Result** $= 1$ **and** $i = 2\}$
            $\Rightarrow$ [*Weak Rule*]
            $\{$ **Result** $= ((i-1)*i)/2)$ **and** $n+1 \geq i$ **and** $i > 1 \}$
        **invariant**
            **Result** $= ((i-1)*i)/2)$ **and** $n+1 \geq i$ **and** $i > 1$
        **until**
            $i = n+1$
        **loop**
            $\{ i \neq n+1$ **and Result** $= ((i-1)*i)/2)$ **and** $n+1 \geq i$ **and** $i > 1 \}$
            $\Rightarrow$ [*Strength Rule*]
            $\{ i \neq n+1$ **and Result** $+ i = (i*(i+1))/2)$ **and** $n+1 \geq i$ **and** $i > 1 \}$
            **Result** $:=$ **Result**$+i$
            $\{ i \neq n+1$ **and Result** $= (i*(i+1))/2)$ **and** $n+1 \geq i$ **and** $i > 1 \}$
            $i := i+1$
            $\{$ **Result** $= ((i-1)*i)/2)$ **and** $n+1 \geq i$ **and** $i > 1 \}$
        **end**
    **ensure**
        **Result** $= (n*(n+1))/2$
    **end**
```

Fig. 3.5: Proof of the *sum* Function

$$\tau(\mathit{Current}) \prec \mathit{MATH} \;\wedge\; n > 1 \;\Rightarrow \mathit{false}$$

Then, applying the strength rule we show:

$$\frac{\{\ \mathit{false}\ \}\quad \mathit{MATH{:}sum}\quad \{\ \mathit{Result} = (n*(n+1))/2\ ,\ \mathit{false}\ \}}{\{\ \tau(\mathit{Current}) \prec \mathit{MATH}\ \wedge\ n > 1\ \}\quad \mathit{MATH{:}sum}\quad \{\ \mathit{Result} = (n*(n+1))/2\ ,\ \mathit{false}\ \}}$$

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

$$\frac{\begin{array}{ccc}\{\ \tau(\mathit{Current}) \prec \mathit{MATH}\ \wedge\ n > 1\ \} & \mathit{MATH{:}sum} & \{\ \mathit{Result} = (n*(n+1))/2\ ,\ \mathit{false}\ \}\\ \{\ \tau(\mathit{Current}) = \mathit{MATH}\ \wedge\ n > 1\ \} & \mathit{impl(MATH,\ sum)} & \{\ \mathit{Result} = (n*(n+1))/2\ ,\ \mathit{false}\ \}\end{array}}{\{\ \tau(\mathit{Current}) \preceq \mathit{MATH}\ \wedge\ n > 1\ \}\quad \mathit{MATH{:}sum}\quad \{\ \mathit{Result} = (n*(n+1))/2\ ,\ \mathit{false}\ \}}$$

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 `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:

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.

$$\begin{array}{lcl} State & \equiv & Local \times Heap \\ Local & \equiv & VarId \cup \{Current, p, Result, Retry\} \rightarrow Value \cup \{undef\} \\ Heap & \equiv & \{\$\} \rightarrow ObjectStore \end{array}$$

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:

$$\langle \sigma, S \rangle \rightarrow \sigma', \chi$$

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 s_1 terminates normally, then the `rescue` clause is not executed, and the returned state is the one produced by s_1 (rule 3.7). If s_1 terminates with an exception, and s_2 triggers another exception, the `rescue` terminates in an exception returning the state produced by s_2 (rule 3.8). If s_1 triggers an exception and s_2 terminates normally, but the *Retry* variable is *false*, then the `rescue` terminates with an exception returning the state produced by s_2 (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 $s_2; s_1$ until s_1 terminates normally or *Retry* is *false*.

$$\frac{\langle \sigma, s_1 \rangle \to \sigma', \mathit{normal}}{\langle \sigma, s_1 \ \texttt{rescue} \ s_2 \rangle \to \sigma', \mathit{normal}} \tag{3.7}$$

$$\frac{\langle \sigma, s_1 \rangle \to \sigma', \mathit{exc} \qquad \langle \sigma', s_2 \rangle \to \sigma'', \mathit{exc}}{\langle \sigma, s_1 \ \texttt{rescue} \ s_2 \rangle \to \sigma'', \mathit{exc}} \tag{3.8}$$

$$\frac{\langle \sigma, s_1 \rangle \to \sigma', \mathit{exc} \qquad \langle \sigma', s_2 \rangle \to \sigma'', \mathit{normal} \qquad \neg\sigma''(\mathit{Retry})}{\langle \sigma, s_1 \ \texttt{rescue} \ s_2 \rangle \to \sigma'', \mathit{exc}} \tag{3.9}$$

$$\frac{\begin{array}{c}\langle \sigma, s_1 \rangle \to \sigma', \mathit{exc} \qquad \langle \sigma', s_2 \rangle \to \sigma'', \mathit{normal} \qquad \sigma''(\mathit{Retry}) \\ \langle \sigma'', s_1 \ \texttt{rescue} \ s_2 \rangle \to \sigma''', \chi\end{array}}{\langle \sigma, s_1 \ \texttt{rescue} \ s_2 \rangle \to \sigma''', \chi} \tag{3.10}$$

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:

```
factorial (i: INTEGER): INTEGER
        -- A once factorial function.
  require
      i>=0
  once
      if i<=1 then
           Result := 1
      else
          Result := i
          Result := Result * factorial (i-1)
      end
  end
```

This example is a typical factorial function but it is also a once function, and the assignment *Result* := $i * \mathit{factorial}(i-1)$ is split into two separate assignments. If one invokes *factorial*(3) we observe that the returned result is 9. The reason is that the first invocation, *factorial*(3), assigns 3 to *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 *factorial*(3) is $3*3$. If we did not split the assignment, the result would be 0 because *factorial*(2) would return the result obtained so far which is the default value of *Result*. This corresponds to a semantics where recursive calls are replaced by *Result*.

To be able to develop a sound semantics for once functions, we need to consider all the possible cases described above. To fulfil 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

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

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

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

$$\frac{\begin{array}{c} T@m = \mathit{impl}(\tau(\sigma(y)), m) \quad T@m \text{ is a once routine} \\ \sigma(T@m\_done) = \mathit{true} \\ \sigma(T@m\_exc) = \mathit{true} \end{array}}{\langle \sigma, x := y.S{:}m(e)\rangle \rightarrow \sigma, \mathit{exc}} \tag{3.14}$$

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 *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:

$$\frac{\sigma(e) = \mathit{exc}}{\langle \sigma, \mathit{Result} := e\rangle \rightarrow \sigma, \mathit{exc}} \tag{3.15}$$

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

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                              class  B
    feature m do ... end                  feature m do ... end
end                                   end


class  C
inherit  A
         B  rename m as n end
end


                                      class  E
class  D                              inherit  C rename m as m2 end
inherit  C  redefine m end
    feature m do ... end              end
end
```

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

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

impl(A,m)	= A@m
impl(B,m)	= B@m
impl(C,m)	= A@m
impl(C,n)	= B@m

impl(D,m)	= D@m
impl(D,n)	= B@m
impl(E,m)	= undefined
impl(E,m2)	= A@m
impl(E,n)	= B@m

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 *env* (the list of classes that defines the program), a type t, and a routine r, *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 *impl* function is defined as follows:

$$\begin{aligned}
&\mathit{impl} : \ \mathit{ClassDeclaration\ list} \times \ \mathit{Type} \times \mathit{RoutineId} \rightarrow \mathit{RoutineImp}\\
&\quad \mathit{impl\ env\ t\ rId} \ = \textbf{if}\ (\mathit{defined\ t\ rId})\ \textbf{then}\ t@\mathit{rId}\\
&\qquad\qquad\qquad\qquad \textbf{else}\ (\mathit{implementation\ env}\ (\mathit{list\_inherits\ env\ t\ rId}))
\end{aligned}$$

The *impl* function is generalized using the function *implementation*. This function takes a list of types and routines. The routine is used to handle renaming. Its definition is as follows:

$$\begin{aligned}
&\mathit{implementation} : \ \mathit{ClassDeclaration\ list} \times ((\mathit{Type} \times \mathit{RoutineId})\ \mathit{list}) \ \rightarrow \mathit{RoutineImp}\\
&\quad \mathit{implementation\ env}\ (t,\ \mathit{rId})\#\mathit{xs} \ = \textbf{if}\ (\mathit{deep\_defined\ env\ t\ rId})\ \textbf{then}\\
&\qquad\qquad\qquad\qquad\qquad (\mathit{impl\ env\ t\ rId})\\
&\qquad\qquad\qquad\qquad \textbf{else}\ (\mathit{implementation\ env\ xs})
\end{aligned}$$

The function *deep_defined* yields *true* if only if given a class declaration list *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 *deep_defined_list* which takes a list of types and routines to handle redefinition. The definitions are as follows:

$$\begin{aligned}
&\mathit{deep\_defined} : \ \mathit{ClassDeclaration\ list} \times ((\mathit{Type} \times \mathit{RoutineId})\ \mathit{list}) \ \rightarrow \mathit{Bool}\\
&\quad \mathit{deep\_defined\ env\ cDecl\ rId} \ = \mathit{undefined}\\
&\quad \mathit{deep\_defined\ env\ cDecl\ rId} \ = \textbf{if}\ (\mathit{defined\ cDecl\ rId})\ \textbf{then}\ \mathit{true}\\
&\qquad\qquad\qquad \textbf{else}\ (\mathit{deep\_defined\_list\ env}\ (\mathit{list\_inherits\ env\ cDecl\ rId}))
\end{aligned}$$

$$\begin{aligned}
&\mathit{deep\_defined\_list} : \ \mathit{ClassDeclaration\ list} \times ((\mathit{Type} \times \mathit{RoutineId})\ \mathit{list}) \ \rightarrow \mathit{Bool}\\
&\quad \mathit{deep\_defined\_list\ env}\ [\,] \ = \mathit{false}\\
&\quad \mathit{deep\_defined\_list\ env}\ (t,\ \mathit{rId})\#\mathit{xs} \ = (\mathit{deep\_defined\ env\ t\ rId}) \vee\\
&\qquad\qquad\qquad (\mathit{deep\_defined\_listenv\ xs})
\end{aligned}$$

Given a type t, and a routine r, the function *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:

$$\begin{array}{ll} \multicolumn{2}{l}{\mathit{list\_inherits} : \mathit{ClassDeclaration\ list} \times \mathit{Type} \times \mathit{RoutineId} \rightarrow (\mathit{Type} \times \mathit{RoutineId})\ \mathit{list}} \\ \quad \mathit{list\_inherits}\ [\,]\ t\ \mathit{rId} & = [\,] \\ \quad \mathit{list\_inherits}\ \mathit{env}\ t\ \mathit{rId} & = (\mathit{list\_inh}\ \mathit{env}\ (\mathit{parents}\ t)\ \mathit{rId}) \end{array}$$

Given a list of class declarations *env*, an inheritance clause *inh*, and a routine r, the function *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:

```
$\mathit{list\_inh} : \mathit{ClassDeclaration\ list} \rightarrow \mathit{Inheritance} \rightarrow \mathit{RoutineId} \rightarrow$
    $((\mathit{ClassDeclaration} \times \mathit{RoutineId})\ \mathit{list})$
$\mathit{list\_inh}\ \mathit{env}\ [\,]\ \mathit{rId} = [\,]$
$\mathit{list\_inh}\ \mathit{env}\ ((t1,\ (\mathit{undef}, \mathit{redef}, \mathit{rename}))\#\mathit{xs})\ \mathit{rId} =$
        if $(\mathit{isUndefined}\ \mathit{undef}\ \mathit{rId})$ then
            $(\mathit{listInh}\ \mathit{env}\ \mathit{xs}\ \mathit{rId})$
        else $(\mathit{renamed\_type}\ \mathit{env}\ t1\ \mathit{rename}\ \mathit{rId})\#(\mathit{list\_inh}\ \mathit{env}\ \mathit{xs}\ \mathit{rId})$
```

where the function *is_undefined* yields *true* if the routine is undefined in the inheritance clause, and the function *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 `rescue` clauses and `retry` variable, we have to extend the signatures of contracts. Preconditions and postconditions of Hoare triples are formulas over $\Sigma\ \cup\ \{\mathit{Current}, p, \mathit{Result}, \mathit{Retry}\}\ \cup\ \mathit{Var}(r)$ where r is a routine, and $\mathit{Var}(r)$ denotes the set of local variables of r. In the case of Eiffel, Σ also contains the special variables for once routines. Note that we assume $\mathit{Var}(r)$ does not include the *Result* variable and the *Retry* variable, it only includes the local variables declared by the programmer. Routine preconditions are formulas over $\Sigma \cup \{\mathit{Current}, p, \$\}$, routine postconditions for normal termination are formulas over $\Sigma\ \cup\ \{\mathit{Result}, \$\}$, and routine exceptional postconditions are formulas over $\Sigma\ \cup\ \$$

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:

$$\frac{\begin{array}{c} P \ \Rightarrow \ I_r \\ \mathcal{A} \vdash \{\ I_r\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \} \\ \mathcal{A} \vdash \{\ Q_e\ \}\ \ s_2\ \ \{\ (Retry \Rightarrow I_r)\ \wedge\ (\neg Retry \Rightarrow R_e)\ ,\ R_e\ \} \end{array}}{\mathcal{A} \vdash \{\ P\ \}\ \ s_1\ \texttt{rescue}\ s_2\ \ \{\ Q_n\ ,\ R_e\ \}}$$

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:

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

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

$$Q'_n \ \equiv \left\{ \begin{array}{l} T@m\_done\ \wedge\ \neg T@m\_exc \wedge \\ (Q_n \vee (\ P''\ \wedge\ Result = T\_M\_RES\ \wedge\ T@m\_result = T\_M\_RES\ )) \end{array} \right\}$$

$$Q'_e \ \equiv \{\ \ T@m\_done\ \wedge\ T@m\_exc\ \wedge\ (Q_e\ \vee P''')\ \ \}$$

The rule for once functions is defined as follows:

$$\frac{\begin{array}{l} \mathcal{A}, \{P\}\ \ T@m\ \{Q'_n,\ Q'_e\} \vdash \\ \\ \{\ P'[false/T@m\_done] \wedge\ T@m\_done\ \}\ \ body(T@m)\ \ \{\ Q_n\ ,\ Q_e\ \} \end{array}}{\mathcal{A} \vdash \{\ P\ \}\ \ T@m\ \ \{\ Q'_n\ ,\ Q'_e\ \}}$$

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, $Q_e \vee P'''$ 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:

$$\frac{T@m \ is \ a \ once \ routine}{\vdash \left\{ \begin{array}{l} (safe(e) \ \wedge \ P[e/Result, e/T@m\_result]) \ \vee \\ (\neg safe(e) \ \wedge \ Q_e) \end{array} \right\} \ Result \ := \ e \ \ \{ \ P \ , \ Q_e \ \}}$$

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:

```
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:

$$\left\{ \ \textit{true} \ \right\} \ \ \textit{MATH:safe\_division} \ \ \left\{ \ Q \ , \ \textit{false} \ \right\}$$

where

$$Q \equiv (y = 0 \Rightarrow \textit{Result} = x) \ \wedge \ (y/ = 0 \Rightarrow \textit{Result} = x//y)$$

Figure 3.9 presents the proof for the body of the routine *safe_division*. To verify this, we first apply the rescue rule (presented on Section 3.2.3). The *retry invariant* is:

$$(y \neq 0 \wedge z = 0) \vee (y = 0 \wedge (z = 1 \vee z = 0))$$

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

```
$\textit{safe\_division}$ ($x, y$: $\textit{INTEGER}$): $\textit{INTEGER}$
    local
        $z$: $\textit{INTEGER}$
    do
        $\{ \ (y \neq 0 \wedge z = 0) \vee (y = 0 \wedge (z = 1 \vee z = 0)) \ \}$
        Result := $x$ // ($y$+$z$)
        $\left\{ \begin{pmatrix} (y = 0 \Rightarrow \textit{Result} = x) \wedge \\ (y \neq 0 \Rightarrow \textit{Result} = x/y) \end{pmatrix}, (y = 0 \wedge z = 0) \right\}$
    ensure
        $\textit{zero}$: $y = 0$ implies Result $= x$
        $\textit{not\_zero}$: $y$ /= 0 implies Result $= x$ // $y$
    rescue
        $\{ \ y = 0 \wedge z = 0 \ \}$
        $z$ := 1
        $\{ \ (y = 0 \wedge z = 1), \ \textit{false} \ \}$
        Retry := true
        $\{ ( \ \textit{Retry} \wedge (y = 0 \wedge z = 1) \ ) , \ \textit{false} \}$
    end
```

Fig. 3.9: Example of an Eiffel Source Proof.

$$\left\{ \ \textit{true} \ \right\} \ \ \textit{body(MATH@safe\_division)} \ \ \left\{ \ Q \ , \ \textit{false} \ \right\}$$

Similar to the example of the function *sum* (presented on Section 3.1.5), we can show that the routine implementation *MATH@safe_division* satisfies its specification applying the routine implementation rule:

$$\frac{\{\ \mathit{true}\ \}\ \ \mathit{body(MATH@safe\_division)}\ \ \{\ Q\ ,\ \mathit{false}\ \}}{\{\ \mathit{true}\ \}\ \ \mathit{MATH@safe\_division}\ \ \{\ Q\ ,\ \mathit{false}\ \}}$$

Since *safe_division* is implemented in the class *MATH*, we have:

$$\mathit{impl(MATH,\ safe\_division)}{=}\mathit{MATH@safe\_division}$$

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

$$\frac{\begin{array}{lcl}\{\ \tau(\mathit{Current}) \prec \mathit{MATH}\ \wedge\ \mathit{true}\ \} & \mathit{MATH{:}safe\_division} & \{\ Q\ ,\ \mathit{false}\ \} \\ \{\ \tau(\mathit{Current}) = \mathit{MATH}\ \wedge\ \mathit{true}\ \} & \mathit{impl(MATH,\ safe\_division)} & \{\ Q\ ,\ \mathit{false}\ \}\end{array}}{\{\ \tau(\mathit{Current}) \preceq \mathit{MATH}\ \wedge\ \mathit{true}\ \}\ \ \mathit{MATH{:}safe\_division}\ \ \{\ Q\ ,\ \mathit{false}\ \}}$$

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 σ and σ' are states, S is an instruction, and χ is the current status of the program. The value of χ can be either the constant *normal*, or *exc*, or *break*. The transition $\langle \sigma, S \rangle \rightarrow \sigma', \mathit{normal}$ expresses that executing the instruction S in the state σ terminates normally in the state σ'. The transition $\langle \sigma, S \rangle \rightarrow \sigma', \mathit{exc}$ expresses that executing the instruction S in the state σ terminates with an exception in the state σ'. The transition

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*@*AttributeId*
	|	*VarId*.*Type*@*AttributeId* := *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.

$\langle \sigma, S \rangle \rightarrow \sigma', \mathit{break}$ expresses that executing the instruction S in the state σ terminates with the execution of a break instruction in the state σ'.

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 χ, 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 χ to break without modifying the state σ, 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 σ. 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 σ', 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 σ'; 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.

Compound

$$\frac{\langle \sigma, s_1\rangle \to \sigma', \mathit{exc}}{\langle \sigma, s_1; s_2\rangle \to \sigma', \mathit{exc}}\,(3.11.1) \qquad \frac{\langle \sigma, s_1\rangle \to \sigma', \mathit{break}}{\langle \sigma, s_1; s_2\rangle \to \sigma', \mathit{break}}\,(3.11.2)$$

$$\frac{\langle \sigma, s_1\rangle \to \sigma', \mathit{normal} \qquad \langle \sigma', s_2\rangle \to \sigma'', \chi}{\langle \sigma, s_1; s_2\rangle \to \sigma'', \chi}\,(3.11.3)$$

Break Instruction

$$\frac{}{\langle \sigma, \texttt{break}\rangle \to \sigma, \mathit{break}}\,(3.11.4)$$

While Instruction

$$\frac{\sigma(e) = \mathit{false}}{\langle \sigma, \texttt{while}\ (e)\ s_1\rangle \to \sigma, \mathit{normal}}\,(3.11.5) \qquad \frac{\sigma(e) = \mathit{true} \qquad \langle \sigma, s_1\rangle \to \sigma', \mathit{exc}}{\langle \sigma, \texttt{while}\ (e)\ s_1\rangle \to \sigma', \mathit{exc}}\,(3.11.6)$$

$$\frac{\sigma(e) = \mathit{true} \qquad \langle \sigma, s_1\rangle \to \sigma', \mathit{break}}{\langle \sigma, \texttt{while}\ (e)\ s_1\rangle \to \sigma', \mathit{normal}}\,(3.11.7) \qquad \frac{\begin{array}{c}\sigma(e) = \mathit{true} \qquad \langle \sigma, s_1\rangle \to \sigma', \mathit{normal}\\ \langle \sigma', \texttt{while}\ (e)\ s_1\rangle \to \sigma'', \chi\end{array}}{\langle \sigma, \texttt{while}\ (e)\ s_1\rangle \to \sigma'', \chi}\,(3.11.8)$$

Fig. 3.11: Operational Semantics for Compound, `while`, and `break` Instructions in Java

Exception Handling

The Java exception handling mechanism is different to the Eiffel mechanism. This difference does not only lie in `try-catch` instructions but also it lies in the fact that Java supports exception types. The semantics of `try` s_1 `catch` (T e) s_2 expresses that if the instruction s_1 triggers an exception, then exception is caught if the type of the exception is a subtype of T, otherwise the exception is propagated.

Throw Instruction

$$\frac{}{\langle \sigma, \texttt{throw}\ e \rangle \to \sigma[excV := e], exc} (3.12.1)$$

Try-Catch Instruction

$$\frac{\langle \sigma, s_1 \rangle \to \sigma', normal}{\langle \sigma, \texttt{try}\ s_1\ \texttt{catch}\ (T\ e)\ s_2 \rangle \to \sigma', normal} (3.12.2)$$

$$\frac{\langle \sigma, s_1 \rangle \to \sigma', exc \quad \tau(\sigma'(excV)) \npreceq T}{\langle \sigma, \texttt{try}\ s_1\ \texttt{catch}\ (T\ e)\ s_2 \rangle \to \sigma', exc} (3.12.3)$$

$$\frac{\langle \sigma, s_1 \rangle \to \sigma', exc \quad \tau(\sigma'(excV)) \preceq T \quad \langle \sigma', s_2 \rangle \to \sigma'', \chi}{\langle \sigma, \texttt{try}\ s_1\ \texttt{catch}\ (T\ e)\ s_2 \rangle \to \sigma'', \chi} (3.12.4)$$

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 $\tau(excV) \npreceq T$), 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++;
          break;
       }
    }
    b++;
}
```

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):

		`finally`		
		normal	**break**	**exc₂**
`try`	**normal**	normal	break	*exc₂*
	break	break	break	*exc₂*
	exc₁	*exc₁*	break	*exc₂*

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

$$\frac{\langle \sigma, s_1 \rangle \to \sigma', \chi \quad \langle \sigma', s_2 \rangle \to \sigma'', normal}{\langle \sigma, \texttt{try } s_1 \texttt{ finally } s_2 \rangle \to \sigma'', \chi} \tag{3.17}$$

$$\frac{\langle \sigma, s_1 \rangle \to \sigma', \chi \quad \langle \sigma', s_2 \rangle \to \sigma'', break}{\langle \sigma, \texttt{try } s_1 \texttt{ finally } s_2 \rangle \to \sigma'', break} \tag{3.18}$$

$$\frac{\langle \sigma, s_1 \rangle \to \sigma', \chi \quad \langle \sigma', s_2 \rangle \to \sigma'', exc}{\langle \sigma, \texttt{try } s_1 \texttt{ finally } s_2 \rangle \to \sigma'', exc} \tag{3.19}$$

In these rules, the state is produced by the execution of the instruction s_2 starting with the sate 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*.

3.3.3 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 expresses 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 $(I \wedge \neg e) \vee\ Q_b$ 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

Compositional Rule:

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

While Rule:

$$\frac{\left\{\ e\ \wedge\ I\ \right\}\ s_1\ \left\{\ I, Q_b, R_e\ \right\}}{\left\{\ I\ \right\}\ \texttt{while}\ (e)\ s_1\ \left\{\ ((I \wedge \neg e) \vee\ Q_b), \mathit{false}, R_e\ \right\}}$$

Break Axiom:

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

Throw Axiom:

$$\frac{}{\left\{\ P[e/\mathit{excV}]\ \right\}\ \texttt{throw}\ e\ \left\{\ \mathit{false}, \mathit{false}, P\ \right\}}$$

try-catch Rule:

$$\frac{\left\{\ P\ \right\}\ s_1\ \left\{\ Q_n, Q_b, Q\ \right\} \qquad \left\{\ Q'_e[e/\mathit{excV}]\ \right\}\ s_2\ \left\{\ Q_n, Q_b, R_e\ \right\}}{\left\{\ P\ \right\}\ \texttt{try}\ s_1\ \texttt{catch}\ (T\ \ e)\ s_2\ \left\{\ Q_n, Q_b, R\ \right\}}$$

where

$Q \equiv (\ (Q''_e\ \wedge\ \tau(\mathit{excV}) \not\preceq T) \vee (Q'_e\ \wedge\ \tau(\mathit{excV}) \preceq T)\ )$

$R \equiv (R_e\ \vee\ (Q''_e\ \wedge\ \tau(\mathit{excV}) \not\preceq T)\ )$

try-finally Rule:

$$\frac{\left\{\ P\ \right\}\ s_1\ \left\{\ Q_n, Q_b, Q_e\ \right\} \qquad \left\{\ Q\ \right\}\ s_2\ \left\{\ R, R'_b, R'_e\ \right\}}{\left\{\ P\ \right\}\ \texttt{try}\ s_1\ \texttt{finally}\ s_2\ \left\{\ R'_n, R'_b, R'_e\ \right\}}$$

where

$$Q \equiv \left(\begin{array}{l} (Q_n \wedge \mathcal{X}\mathit{Tmp} = \mathit{normal})\ \vee (Q_b \wedge \mathcal{X}\mathit{Tmp} = \mathit{break}) \vee \\ \left(\ Q_e[\mathit{eTmp}/\mathit{excV}] \wedge \mathcal{X}\mathit{Tmp} = \mathit{exc} \wedge \mathit{eTmp} = \mathit{excV}\ \right) \end{array}\right)$$

$$R \equiv \left(\begin{array}{l} (R'_n \wedge \mathcal{X}\mathit{Tmp} = \mathit{normal})\ \vee\ (R'_b \wedge \mathcal{X}\mathit{Tmp} = \mathit{break}) \vee \\ (R'_e[\mathit{eTmp}/\mathit{excV}] \wedge \mathcal{X}\mathit{Tmp} = \mathit{exc}) \end{array}\right)$$

Fig. 3.13: Rules for Composition, `while`, `break`, `try-catch`, and `throw` Instructions.

postcondition is $Q''_e \ \wedge \ \tau(excV) \not\preceq T$, where τ yields the runtime type of an object, $excV$ is a variable that stores the current exception, and $\preceq$ denotes subtyping; (3) s_1 throws an exception and the exception is caught. In the postcondition of s_1, $Q'_e \ \wedge \ \tau(excV) \preceq T$ 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 `break`.

To define the rule for `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 `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: *normal*, *break*, and *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 *break*.

3.3.4 Example

Figure 3.14 presents an example of a proof of a Java function using `while`, `break`, and `try-finally` instructions. In the *foo* function, an exception is thrown in the `try` block with precondition $b = 1$. The `finally` block increases b and then executes a `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 *foo* satisfies the following specification:

$$\left\{ \ true \ \right\} \quad MATH{:}foo \quad \left\{ \ b = 3 \ , \ false \ \right\}$$

To prove that the body of the routine *foo* satisfies the following specification:

$$\left\{ \ true \ \right\} \quad body(MATH@foo) \quad \left\{ \ b = 3, false, false \ \right\}$$

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

Applying the routine implementation rule, we can connect the specification of the body of *foo* and the method specification. The body of *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:

$$\frac{\left\{ \ true \ \right\} \quad body(MATH@foo) \quad \left\{ \ b = 3, false, false \ \right\}}{\left\{ \ true \ \right\} \quad MATH@foo \quad \left\{ \ b = 3 \ , \ false \ \right\}}$$

```
**void** *foo* (**int** $b$) {
    $\left\{ \ true \ \right\}$
    $b = 1;$
    $\left\{ \ b = 1, \ false, \ false \ \right\}$
    **while** (**true**) {
        $\left\{ \ b = 1, \ false, \ false \ \right\}$
        **try** {
            $\left\{ \ b = 1, \ false, \ false \ \right\}$
            **throw new** *Exception*();
            $\left\{ \ false, \ false, \ b = 1 \ \right\}$
        }
        **finally** {
            $\left\{ \ b = 1 \wedge Xtmp = exc \ \right\}$
            $b \ = \ b{+}1;$
            $\left\{ \ b = 2 \wedge Xtmp = exc, \ false, \ false \ \right\}$
            **break**;
            $\left\{ \ false, \ b = 2 \wedge Xtmp = exc, \ false \ \right\}$
        }
        $\left\{ \ false, \ b = 2, \ false \ \right\}$
    }
    $\left\{ \ b = 2, \ false, \ false \ \right\}$
    $b \ = \ b{+}1;$
    $\left\{ \ b = 3, \ false, \ false \ \right\}$
}
```

Fig. 3.14: Example of a Proof of a Java Function using `while`, `break`, and `try-finally` Instructions.

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:

$$\tau(\mathit{Current}) \prec \mathit{MATH} \ \wedge \ n > 1 \ \Rightarrow \mathit{false}$$

Then, applying the strength rule we show:

$$\frac{\{ \ \mathit{false} \ \} \quad \mathit{MATH{:}foo} \quad \{ \ b = 3 \ , \ \mathit{false} \ \}}{\{ \ \tau(\mathit{Current}) \prec \mathit{MATH} \ \wedge \ \mathit{true} \ \} \quad \mathit{MATH{:}foo} \quad \{ \ b = 3 \ , \ \mathit{false} \ \}}$$

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:

$$\frac{\begin{array}{ccc} \{ \ \tau(\mathit{Current}) \prec \mathit{MATH} \ \wedge \ \mathit{true} \ \} & \mathit{MATH{:}foo} & \{ \ b = 3 \ , \ \mathit{false} \ \} \\ \{ \ \tau(\mathit{Current}) = \mathit{MATH} \ \wedge \ \mathit{true} \ \} & \mathit{impl(MATH, foo)} & \{ \ b = 3 \ , \ \mathit{false} \ \} \end{array}}{\{ \ \tau(\mathit{Current}) \preceq \mathit{MATH} \ \wedge \ \mathit{true} \ \} \quad \mathit{MATH{:}foo} \quad \{ \ b = 3 \ , \ \mathit{false} \ \}}$$

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 $\{ \ P \ \} \quad s \quad \{ \ Q_n \ , \ Q_e \ \}$. 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* $\models \{ \ P \ \} \quad s \quad \{ \ Q_n \ , \ Q_e \ \}$ *if and only if:*
for all $\sigma \models P : \langle \sigma, s \rangle \rightarrow \sigma', \chi$ *then*

- $\chi = \mathit{normal} \ \Rightarrow \sigma' \models Q_n$, *and*
- $\chi = \mathit{exc} \ \Rightarrow \sigma' \models Q_e$

Theorem 1 (Soundness Theorem)**.**

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

Theorem 2 (Completeness Theorem)**.**

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

3.5 Related Work

Abadi and Leino [1] 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-stype 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 [4] 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 Bierman [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 refiniment 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 `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 programing 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 $\mathit{factorial}(n)$ 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:

US_map. *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, $a.call(t, arg)$[1], 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.

[1]To simplify the notation, we use a slight variant of the Eiffel.

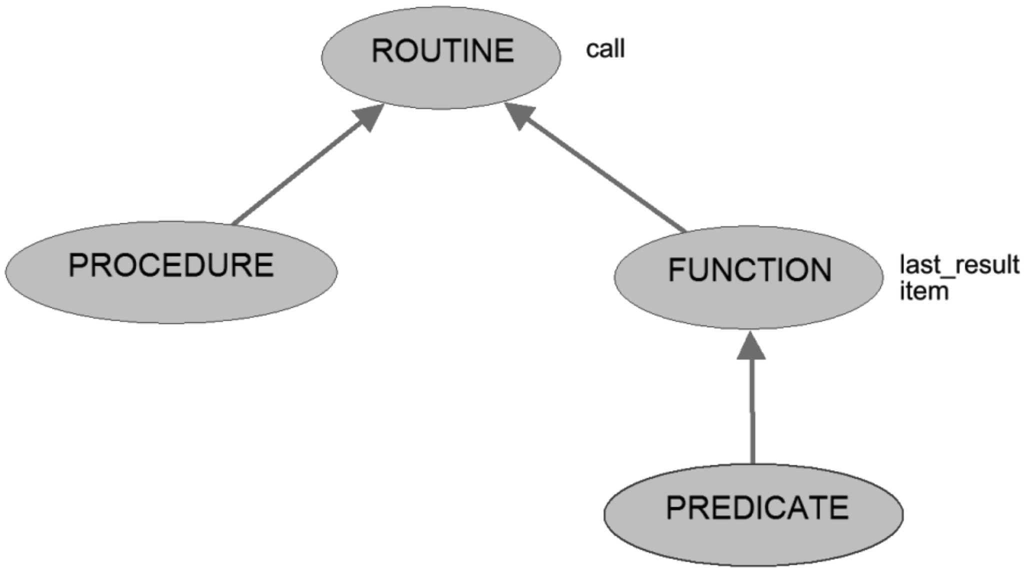

Fig. 4.1: Kernel Library Classes for Agents.

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[2]. *PROCEDURE* and *PREDICATE* have the same generic parameters except the last.

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 function 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

[2]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 **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. [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 *FORMATTER* models paragraph formatting with two alignment routines. The class *PARAGRAPH* includes a procedure to format the current paragraph. For illustration purposes, the routines *align_left* and *align_right* require that the paragraph is not left aligned and not right aligned, respectively. The routines *left_aligned* and *right_aligned* are pure routines (side effect free) defined in the class *PARAGRAPH*, and return *true* if the paragraph is left aligned or right aligned, respectively. These classes are implemented in Eiffel as follows:

```
class FORMATTER

  align_left (p: PARAGRAPH)                align_right (p: PARAGRAPH)
    require                                  require
        not p. left_aligned                      not p. right_aligned
    do                                       do
      ... Operations on p ...                 ... Operations on p ...
    ensure                                   ensure
        p. left_aligned                          p. right_aligned
    end                                      end
end
```

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

The signature *proc: PROCEDURE [FORMATTER, PARAGRAPH]*[3] declares a procedure *proc* with two open arguments (the target of type *FORMATTER* and a parameter of type *PARAGRAPH*). Open arguments are the arguments provided in the invocation of the agent. An example of the use of the *format* routine is shown in the routine *apply_align_left*. This routine is implemented as follows:

```
apply_align_left  (f: FORMATTER; p: PARAGRAPH)
    require
        not p. left_aligned
    do
        p. format (agent { FORMATTER }.align_left , f )
    ensure
        p. left_aligned
    end
```

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

4.2.2 Multi-Level Undo-Redo

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

```
undo_all ( history_list : LIST [COMMAND] )
 do
    history_list . do_if(agent {COMMAND}.cancel, agent {COMMAND}.cancelable)
 end
```

The routine *cancelable* returns *true* if the command can be canceled; in other words it satisfies the precondition of *cancel*. The iteration routine *do_if* is available to all list

[3]To simplify the notation, we do not use *TUPLE* in the declaration of the agent. Thus, we write *PROCEDURE [FORMATTER, PARAGRAPH]* instead of *PROCEDURE [FORMATTER, TUPLE [FORMATTER, PARAGRAPH]*.

classes through their declaration in the ancestor class *LINEAR*, where its implementation is:

```
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
    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[4].

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 *PROCEDURE[ANY;TAPE]*[5] 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 t defined in ***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 *precondition* and *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 *format* with a contract:

[5]This is a simplification of the declaration in Eiffel. The declaration in Eiffel is *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.

```
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[6] *@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 *@postcondition* takes a second object store to evaluate old expressions. The signatures of these functions are defined as follows[7]:

$$
\begin{aligned}
&@\mathit{precondition} : \mathit{Value} \times \mathit{Value} \times \mathit{Value} \times \mathit{ObjectStore} \rightharpoonup \mathit{Bool}\\
&@\mathit{postcondition} : \mathit{Value} \times \mathit{Value} \times \mathit{Value} \times \mathit{ObjectStore} \times \mathit{ObjectStore} \rightharpoonup \mathit{Bool}
\end{aligned}
$$

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

[6]We use the prefix @ in the mathematical functions to distinguish them from the Eiffel routines.

[7]$\rightharpoonup$ denotes partial functions.

$@pre_{pr} : \mathit{Value} \times \mathit{Value} \times \mathit{ObjectStore} \rightharpoonup \mathit{Bool}$
$@post_{pr} : \mathit{Value} \times \mathit{Value} \times \mathit{ObjectStore} \times \mathit{ObjectStore} \rightharpoonup \mathit{Bool}$

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

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

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 *call* of the class *ROUTINE*, it suffices to rely on the specification of this routine. We assume that the implementation of *call* in *ROUTINE* is correct with respect to this specification. The following axiom defines this assumption:

$$\frac{}{\{\, @\mathit{precondition}(\mathit{Current}, t, p, \$) \,\}\ \mathit{ROUTINE}{:}\mathit{call}(t, p)\ \{\, @\mathit{postcondition}(\mathit{Current}, t, p, \$, \$') \,\}}$$

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

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

$$\{\ @\mathit{precondition}(a, t, p, \$)\ \}\quad a.\mathit{call}(t, p)\quad \{\ @\mathit{postcondition}(a, t, p, \$, \$')\ \}$$

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

Agent Initialization

The agent initialization is treated as a normal assignment, where the routine *pr* is assigned to the agent a.

$$\frac{}{\{\ P[pr/a]\ \}\quad a := \textbf{agent}\ pr\quad \{\ P\ \}}$$

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 *$precondition* takes three values (the agent, the target and the parameter), and the current heap, and yields the evaluation of the agent's precondition. The function *$postcondition* takes a second heap to evaluate old expressions. The signatures of these functions are defined as follows:

$$\$\mathit{precondition} : \mathit{Value} \times \mathit{Value} \times \mathit{Value} \times \mathit{Heap} \rightharpoonup \mathit{Bool}$$
$$\$\mathit{postcondition} : \mathit{Value} \times \mathit{Value} \times \mathit{Value} \times \mathit{Heap} \times \mathit{Heap} \rightharpoonup \mathit{Bool}$$

Initializing Agents

Given the agent initialization $a := \textbf{agent } pr$ where pr is a procedure[8], the methodology generates the following assumptions:

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

where $\$\mathit{pre}_{pr}$ and $\$\mathit{post}_{pr}$ 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, $a.\mathit{call}(t, p)$, first asserts the precondition of the agent, and then assumes its postcondition. The proof obligations are the followings:

```
**assert** $\$\mathit{precondition}(a, t, p, \mathit{Heap})$
$h_0 := \mathit{Heap}$
**havoc** $\mathit{Heap}$
**assume** $\$\mathit{postcondition}(a, t, p, \mathit{Heap}, h_0)$
```

The current heap is denoted by *Heap*. The assignment $h_0 := \mathit{Heap}$ saves the current heap, then h_0 is used to evaluate the postcondition of the agent. The **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 $\$\mathit{precondition}$ and $\$\mathit{postcondition}$ 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 $f : \mathit{Heap} \rightharpoonup X$, 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 $\mathit{Heap} \rightharpoonup X$ are obtained from preconditions and bodies of agents by fixing the target and

[8]The syntax **agent** *pr* is an abbreviation for keeping all arguments open, as in **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 $f : \mathit{Heap} \rightharpoonup X$ and $g : \mathit{Heap} \rightharpoonup Y$, we write the noninterference predicate $f\#g : \mathit{Heap} \rightarrow \mathit{Bool}$ 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 $o, o' \in \mathit{ObjectId}$ objects of class C, we define

$$\begin{array}{l} o\#o' : \mathit{Heap} \rightarrow \mathit{Bool} \\ (o\#o')(h) \triangleq \forall v, v' \in \mathit{Value}, \forall f, f' \in F.\ (f(o,v)\#f'(o',v'))(h) \end{array}$$

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:

$$\$\mathit{precondition}_1 \ \mathit{and}\ \$\mathit{postcondition}_1{}^{9}$$

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

$$\begin{array}{l} \$\mathit{precondition}_1 : \mathit{Value} \times \mathit{Value} \times \mathit{Heap} \rightharpoonup \mathit{Bool} \\ \$\mathit{postcondition}_1 : \mathit{Value} \times \mathit{Value} \times \mathit{Heap} \times \mathit{Heap} \rightharpoonup \mathit{Bool} \end{array}$$

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

$$\begin{array}{l} \$\mathit{precondition}_i : \ \mathit{Value}^{i+i} \times \mathit{Heap} \rightharpoonup \mathit{Bool} \\ \$\mathit{postcondition}_i : \ \mathit{Value}^{i+i} \times \mathit{Heap} \times \mathit{Heap} \rightharpoonup \mathit{Bool} \end{array}$$

can be used to model agents with i open arguments.

[9] 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 $\$\mathit{precondition}_1$ and $\$\mathit{postcondition}_1$. In the following, we present these assumptions for closed target and closed arguments.

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

assume $\forall p : \mathit{ObjectId}; h_1 : \mathit{Heap} : \$\mathit{precondition}_1(a, p, h_1) = \$\mathit{pre}_{\mathit{pr}}(t_1, p, h_1)$
assume $\forall p : \mathit{ObjectId}; h_1, h_2 : \mathit{Heap} : \$\mathit{postcondition}_1(a, p, h_1, h_2) = \$\mathit{post}_{\mathit{pr}}(t_1, p, h_1, h_2)$

These assumptions quantify only over one parameter, p. The target object t_1 is known, and it is used in the function $\$\mathit{pre}_{\mathit{pr}}$. 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 := **agent** $\mathit{pr}(p_1)$ where p_1 is the closed parameter, and pr a procedure, the methodology generates the following assumptions:

assume $\forall t : \mathit{ObjectId}; h_1 : \mathit{Heap} : \$\mathit{precondition}_1(a, t, h_1) = \$\mathit{pre}_{\mathit{pr}}(t, p_1, h_1)$
assume $\forall t : \mathit{ObjectId}; h_1, h_2 : \mathit{Heap} : \$\mathit{postcondition}_1(a, t, h_1, h_2) = \$\mathit{post}_{\mathit{pr}}(t, p_1, h_1, h_2)$

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 $a.\mathit{call}(p)$ with argument p, defines the following proof obligations:

```
**assert** $\$\mathit{precondition}_1(a, p, \mathit{Heap})$
$h_0 := \mathit{Heap}$
**havoc** $\mathit{Heap}$
**assume** $\$\mathit{postcondition}_1(a, p, \mathit{Heap}, h_0)$
```

where *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 $\$\mathit{precondition}_1$ and $\$\mathit{postcondition}_1$ 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.

4.4.3 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 *modifies*. The definition of modifies clauses and routines declarations is the following:

$$
\begin{array}{lcl}
\textit{Modifies\_clause} & ::= & \textit{Modifies\_clause}, \textit{Modifies\_clause} \\
 & & \mid \textit{VarId} \\
 & & \mid \textit{modifies}(\textit{VarId}, \textit{VarId}, \textit{VarId}) \\
\textit{Routine} & ::= & \textit{RoutineId}\ (\textit{VarId} : \textit{Type}) :\ \textit{Type} \\
 & & \quad \texttt{require}\ \textit{BoolExp} \\
 & & \quad \texttt{modify}\ \textit{Modifies\_clause} \\
 & & \quad \texttt{do} \\
 & & \qquad \textit{Instr} \\
 & & \quad \texttt{ensure}\ \textit{BoolExp} \\
 & & \quad \texttt{end}
\end{array}
$$

where *boolExp* are boolean expressions, *RoutineId* routine identifiers, *VarId* variable identifiers, and *Instr* instructions.

Encoding of Modifies Clauses. To encode the modifies clauses, we introduce a function $\$modifies$ 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:

$$\$\textit{modifies} : \textit{Value} \times \textit{Value} \times \textit{Value} \times \textit{Heap} \times \textit{Value} \times \textit{FieldId} \rightharpoonup \textit{Bool}$$

For example, the modifies clause of the routine *format* can be encoded using this function as follows:

$$
\begin{array}{l}
\textbf{ensures}\ \forall o : \textit{ObjectId};\ \textit{fId} : \textit{FieldId} : \\
\quad \textit{not}\ \$\textit{modifies}(\textit{proc}, f, \textit{Current}, \textit{Heap}, o, \textit{fId}) \Rightarrow \textit{Heap}[o, \textit{fId}] = \textit{old}(\textit{Heap})[o, \textit{fId}]
\end{array}
$$

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

Generalizing, modifies clauses are list of application of the function $\$modifies$ and variable identifiers. Given the modifies clause:
$\$\textit{modifies}(a_1, t_1, p_1), ..., \$\textit{modifies}(a_n, t_n, p_n), v_1, ..., v_m$

this clause is encoded as:

$$\textbf{ensures}\ \forall o : \mathit{ObjectId};\ \mathit{fId} : \mathit{FieldId} :$$
$$\left(\begin{array}{l} \mathit{not}\ \$\mathit{modifies}(a_1, t_1, p_1, \mathit{Heap}, o, \mathit{fId}) \\ \wedge \ldots \wedge \\ \mathit{not}\ \$\mathit{modifies}(a_n, t_n, p_n, \mathit{Heap}, o, \mathit{fId}) \\ \wedge\ o! = v_1\ \wedge\ o! = v_m \end{array}\right) \Rightarrow\ \mathit{Heap}[o, \mathit{fId}] = \mathit{old}(\mathit{Heap})[o, \mathit{fId}]$$

Initializing Agents. To solve the frame problem for agents, we need to link the function $\$\mathit{modifies}(\mathit{proc}, t, p)$ with the locations that the routine *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 $\$\mathit{modifies}$, when the agent is initialized. Given a procedure *pr*, the agent initialization $a\ :=$ **agent** *pr* generates the following assumptions:

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

$$\textbf{assume}\ \forall t, p, o : \mathit{ObjectId};\ \mathit{fId} : \mathit{FieldId}; h_1 : \mathit{Heap} :$$
$$\$\mathit{modifies}(a, t, p, h_1, o, \mathit{fId}) = \$\mathit{modifies}_{pr}(t, p, o, \mathit{fId})$$

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

$$\$\mathit{modifies}_{\mathit{align\_left}}(\mathit{Current}, p, o, \mathit{fId}) \triangleq (o = p)$$

Generalizing, the function $\mathit{modifies}_{pr}$ 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 *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 $\mathit{modifies}_1$. 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:

$$\begin{array}{lll} \textit{Modifies\_clause} & ::= & \textit{Modifies\_clause}, \textit{Modifies\_clause} \\ & & \mid \textit{VarId} \\ & & \mid \textit{modifies}(\textit{VarId}, \textit{VarId}, \textit{VarId}) \\ & & \mid \textit{modifies}_1(\textit{VarId}, \textit{VarId}) \end{array}$$

Using the function *modifies*$_1$, the modifies clause of the routine *log* (Section 4.2.3) can be defined as follows:

```
log ( log_file : PROCEDURE [ANY;TAPE]; s: STRING)
    modify
        modifies₁( log_file ,s)
    do
        log_file . call(s)
    end
```

Encoding of Modifies Clauses. The encoding of the modifies clauses follows the same ideas of the above section. We define a function $\$\textit{modifies}_1$ which takes an agent a, its open argument's value, the current heap, an object value o, and a field name *fId*, and yields *true* if the agent a with argument p modifies the field $o.\textit{fId}$. The signature of this function is the following:

$$\$\textit{modifies}_1 : \textit{Value} \times \textit{Value} \times \textit{Heap} \times \textit{Value} \times \textit{FieldId} \rightharpoonup \textit{Bool}$$

For example, the modifies clause of *log* can be encoded as follows:

$$\begin{array}{l} \textbf{free ensures } \forall o : \textit{ObjectId};\ \textit{fId} : \textit{FieldId} : \\ \quad \textit{not } \$\textit{modifies}_1(\textit{log\_file}, s, \textit{Heap}, o, \textit{fId}) \Rightarrow \textit{Heap}[o, \textit{fId}] = \textit{old}(\textit{Heap})[o, \textit{fId}] \end{array}$$

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 := **agent** $t_1.pr$ where t_1 is the closed target, and pr a procedure, the methodology generates the following assumptions:

```
assume $\forall p : \mathit{ObjectId}; h_1 : \mathit{Heap} : \$\mathit{precondition}_1(a, p, h_1) = \$\mathit{pre}_{pr}(t_1, p, h_1)$
assume $\forall p : \mathit{ObjectId}; h_1, h_2 : \mathit{Heap} : \$\mathit{postcondition}_1(a, p, h_1, h_2) = \$\mathit{post}_{pr}(t_1, p, h_1, h_2)$

assume $\forall o, p : \mathit{ObjectId};\ \mathit{fId} : \mathit{FieldId} :$
        $\$\mathit{modifies}_1(a, p, o, \mathit{fId}) = \$\mathit{modifies}_{pr}(t_1, p, o, \mathit{fId})$
```

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:

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

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  $\textbf{assume not } p.\$left\_aligned$
 2  $a := \textbf{agent}\{FORMATTER\}.align\_left$
 3  $\textbf{assume } \forall t_1, p_1 : ObjectId; h : Heap :$
            $\$precondition(a, t_1, p_1, h) = \$pre_{align\_left}(t_1, p_1, h)$
 4  $\textbf{assume } \forall t_1, p_1 : ObjectId; h, h' : Heap :$
            $\$postcondition(a, t_1, p_1, h, h') = \$post_{align\_left}(t_1, p_1, h, h')$
 5  $\textbf{assume } \forall t_1, p_1, o : ObjectId; fId : FieldId; h : Heap :$
            $\$modifies(a, t_1, p_1, h, o, fId) = \$modifies_{align\_left}(t_1, p_1, o, fId)$
 6  $\textbf{assert } \$precondition(a, f, p, Heap)$
 7  $h_0 := Heap$
 8  $\textbf{havoc } Heap$
 9  $\textbf{assume } \$postcondition(a, f, p, Heap, h_0)$
10  $\textbf{assume } \forall o : ObjectId; fId : FieldId :$
            $not\ \$modifies(proc, f, p, Heap, o, fId) \Rightarrow Heap[o, fId] = h_0[o, fId]$
11  $\textbf{assert } p.\$left\_aligned$
12  $\textbf{assert } \forall o : ObjectId; fId : FieldId : o! = p \Rightarrow Heap[o, fId] = h_0[o, fId]$
```

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 $\$pre_{align\_left}$ and $\$post_{align\_left}$ respectively; the modifies clause of *align_left* is denoted by $\$modifies_{align\_left}$. The invocation of the routine *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 $\$pre_{align\_left}$, $\$post_{align\_left}$, and $\$modifies_{align\_left}$ we know:

$$\forall t_1, p_1 : ObjectId; h : Heap : \$pre_{align\_left}(t_1, p_1, h) = \ not\ p_1.\$left\_aligned \tag{4.1}$$

$$\forall t_1, p_1 : ObjectId; h, h' : Heap : \$post_{align\_left}(t_1, p_1, h, h') = \ p_1.\$left\_aligned \tag{4.2}$$

$$\$modifies_{align\_left}(Current, p, o, fId) \ = o! = p \tag{4.3}$$

In particular, $\$pre_{align\_left}(f, p, Heap) = \ not\ p.\$left\_aligned$. 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).

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:

```
do_if (f: PROCEDURE [ANY]; test: FUNCTION [ANY])
  require
      forall 1≤i≤count: test(ith(i)) implies f.precondition(ith(i))
  ensure
      forall 1≤i≤count: ( old test(ith(i)) implies f.postcondition(ith(i)) )
```

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 trigged.

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 [7, 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: PROCEDURE [ANY]; test: FUNCTION [ANY])
  require
      forall 1≤i≤count: test(ith(i)) implies f.precondition(ith(i)) and
      forall 1 ≤ i < j ≤ count: ith(i) # ith(j)
  ensure
      forall 1≤i≤count: ( old test(ith(i)) implies f.postcondition(ith(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 *do_if*. Section 4.4 shows how this example is encoded and automatically proved in Spec#.

Let *do_if_pre* be the precondition of the routine *do_if* and *loop_invariant* be the loop invariant defined as follows:

$$\begin{aligned}
&do\_if\_pre \triangleq\ Inv(\alpha)\ \wedge\ forall\ 1 \le i < j \le count : ith(i)\#ith(j)\ \wedge\\
&\quad forall\ 1 \le i \le count : ((\alpha_i.\$test \Rightarrow \$precondition(f, \alpha_i, Heap)) \wedge (\alpha_i.\$test\ = \beta_i))\\
&loop\_invariant \triangleq\ 1 \le j \le count + 1\ \wedge\ forall\ j \le i < k \le count : ith(i)\#ith(k)\ \wedge\\
&\qquad forall\ 1 \le i \le j-1 : (\beta_i \Rightarrow \$postcondition(f, \alpha_i, Heap, h_0)\ \wedge\\
&\qquad forall\ j \le i \le count :\ (\alpha_i.\$test\ \Rightarrow \$precondition(f, \alpha_i, Heap))
\end{aligned}$$

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

Figure 4.3 presents the sketch of the proof of the routine *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.

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 *main*.

The proof for the routine *log* is similar to the proof of the *format* routine. The only change is the use of the function $\$precondtion_1$ which takes only three arguments (the procedure *log_file*, the string s and the heap). The proof obligations are:

```
$do\_if\ (f : PROCEDURE[ANY];\ test : FUNCTION[ANY])$
  do
2   assume $do\_if\_pre$
    $h_0 := Heap$
4   from $start$ until $after$ loop
      if $test(item)$ then
6       assert $\$precondition(f, \alpha_j, Heap)$
        $h_1 := Heap$
8       $f.call(item)$
        assume $\$postcondition(f, \alpha_j, Heap, h_1)$
10    end
      $forth$
12    assert $loop\_invariant$
    end
14  assert $\left( \begin{array}{l} Inv(\alpha) \wedge\ forall\ 1 \leq i < k \leq count : ith(i)\#ith(k)\ \wedge \\ (forall\ 1 \leq i \leq count : (\beta_j \Rightarrow \$postcondition(f, \alpha_j, Heap, h_0)) \end{array} \right)$
  end
```

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

```
$log(log\_file : PROCEDURE[ANY; TAPE];\ s : STRING)$
  1 assume $\$precondition_1(log\_file, s, Heap)$
  2 assert $\$precondition_1(log\_file, s, Heap)$
  3 $log\_file.call(s)$
```

The proof of routine *main* translates the agent initialization in lines 3-5. The function $precondition_1$ 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:

```
$main(c : CLIENT)$
  1 create $t.make$
  2 assert $t.\$is\_loaded$
  3 $a :=$ agent $t.store$
  4 assume $\forall p_1 : ObjectId; h : Heap :$
          $\$precondition_1(a, p_1, h) = \$pre_{store}(a, t, p_1, h)$
  5 assume $\forall p_1 : ObjectId; h, h' : Heap :$
          $\$postcondition_1(a, p_1, h, h') = \$post_{store}(a, t, p_1, h, h')$
  6 assert $\$precondition_1(a,$ "HelloWorld"$, Heap)$
  7 $c.log(a,$ "HelloWorld"$)$
```

	Name	*Classes*	*Routines*	*Agents*	*LOC Eiffel*	*LOC Boogie*	*Time [s]*
1.	Formatter	3	8	2	116	414	1.57
2.	Archiver	4	7	1	119	440	1.58
3.	Command	3	5	2	120	435	1.61
4.	Calculator	3	10	15	243	817	25.14
5.	ATM	4	18	13	486	1968	73.72
6.	Cell / Recell	3	7	4	151	497	1.71
7.	Counter	2	5	2	96	356	1.53
8.	Sequence	5	9	2	200	526	1.78
	Total	27	69	41	1513	5453	108.64

Tab. 4.1: Examples automatically verified by *EVE Proofs*

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`. 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.

Parkinson and Bierman [101, 102] introduce abstract predicates to verify object-oriented programs in separation logic. Abstract predicates are a powerful means to abstract from implementation details and to support information hiding and inheritance. 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 *precondition* and *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 bin_{op} if op is a binary function. We consider the following binary instructions:

    1. add, rem, mul, 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. ceq, clt, 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.

- brtrue l: transfers control to the point l if the top element of the stack is *true* and unconditionally pops it.

- brfalse l: transfers control to the point l if the top element of the stack is *false* and unconditionally pops it.

- br l: transfers control to the point l.

- leave l: exits from .try catch and .try .finally blocks transferring control to the point l.

- endfinally: exits the .finally block transferring control to the end of the block.

- checkcast T: checks whether the top element is of type T or a subtype thereof.

- newobj instance *void Class::.ctor().* T: allocates a new object of type T and pushes it onto the stack.

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

- ldfld F: replaces the top element by its field F (an instance field).

- ldsfld F: replaces the top element by its field F (a static field).

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

- stsfld F: sets the field F (a static field) of the object denoted by the second-topmost element to the top element of the stack, and pops both values.

- ret: returns to caller.
- nop: has no effect.
- .try { seq_1 } catch T { seq_2 }: executes the sequence of CIL instructions seq_1. If any instruction in seq_1 throws an exception of type T, control is transferred to the first instruction in seq_2.
- .try { seq_1 } .finally { seq_2 }: executes the sequence of CIL instructions seq_1, and then it executes the sequence seq_2 (even if an exception is triggered in seq_1).

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 $\{P\}\ m\ \{Q_n,\ Q_e\}$ where Q_n is the postcondition after normal termination, Q_e is the exceptional postcondition, and m is a virtual method $T:m$ or a method implementation $T@m$. 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
    IL012 : ldloc Result
    IL013 : ldloc i
    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 $\{\mathrm{P}\}\ m\ \{Q\}$. We have extended the logic to handle exceptions; for this reason, method specifications have the form $\{\mathrm{P}\}\ m\ \{Q_n,\ Q_e\}$.

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.

5.2.2 Rules

Rules for Instruction Specifications

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

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

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.

I_l	$wp(I_l)$
ldc v	$unshift(E_{l+1}[v/s(0)])$
ldloc x	$unshift(E_{l+1}[x/s(0)])$
stloc x	$(shift(E_{l+1}))[s(0)/x]$
bin_{op}	$(shift(E_{l+1}))[s(1)\ op\ s(0)/s(1)]$
brtrue l'	$(\neg s(0) \Rightarrow shift(E_{l+1})) \wedge (s(0) \Rightarrow shift(E_{l'}))$
brfalse l'	$(s(0) \Rightarrow shift(E_{l+1})) \wedge (\neg s(0) \Rightarrow shift(E_{l'}))$
br l'	$E_{l'}$
leave l'	$E_{l'}$
nop	E_{l+1}
checkcast T	$E_{l+1} \wedge \tau(s(0)) \preceq T$
newobj T	$unshift(E_{l+1}[new(\$, T)/s(0), \$\langle T\rangle/\$]$
ldfld $T@a$	$E_{l+1}[\$(iv(s(0), T@a))/s(0)] \ \wedge\ s(0) \neq null$
stfld $T@a$	$(shift^2(E_{l+1}))[\$\langle iv(s(1), T@a) := s(0)\rangle/\$] \ \wedge\ s(1) \neq null$
ret	$true$

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 $[E_l] : State \times Stack \rightarrow Value$ for s is defined as follows:

$$\begin{aligned}&[s(0)]\ \langle S,(\sigma,v)\rangle\ = v \textit{ and}\\ &[s(i+1)]\ \langle S,(\sigma,v)\rangle = [s(i)]\langle S,\sigma\rangle\end{aligned}$$

The functions *shift* and *unshift* define the substitutions that occur when values are pushed onto and popped from the stack, respectively. Their definitions are the following:

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

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:

$$\frac{\begin{array}{c}P \Rightarrow E_0 \qquad\qquad E_{|body(T@m)|-1} \Rightarrow Q_n\\[4pt] \forall i : \{0,..,\mid body(T@m)\mid -1\} : \mathcal{A} \vdash \{E_i\} i : I_i\end{array}}{\mathcal{A} \vdash \{\ P\ \}\ \ body(T@m)\ \ \{\ Q_n\ ,\ Q_e\ \}}$$

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

$$\frac{\mathcal{A}, \{P\}\ \ T@m\ \ \{Q_n,\ Q_e\} \vdash \left\{\ P\ \right\}\ \ body(T@m)\ \ \left\{\ Q_n\ ,\ Q_e\ \right\}}{\mathcal{A} \vdash \left\{\ P\ \right\}\ \ T@m\ \ \left\{\ Q_n\ ,\ Q_e\ \right\}}$$

Class Rule:

$$\frac{\begin{array}{c}\mathcal{A} \vdash \left\{ \ \tau(Current) = T \ \wedge \ P \ \right\} \ \ impl(T, m) \ \ \left\{ \ Q_n \ , \ Q_e \ \right\} \\ \mathcal{A} \vdash \left\{ \ \tau(Current) \prec T \ \wedge \ P \ \right\} \ \ T{:}m \ \ \left\{ \ Q_n \ , \ Q_e \ \right\}\end{array}}{\mathcal{A} \vdash \left\{ \ \tau(Current) \preceq T \ \wedge \ P \ \right\} \ \ T{:}m \ \ \left\{ \ Q_n \ , \ Q_e \ \right\}}$$

Subtype Rule:

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

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 *false*. In the *assumpt-axiom*, assuming that a triple $\mathbf{A}$ holds, one can conclude the same triple. The *assumpt-intro-axiom* allows us to introduce a triple $\mathbf{A}_0$ in the hypothesis. The *assumpt-elim-axiom* allows eliminating a triple $\mathbf{A}_0$ 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

Assumpt-axiom

$$\frac{}{\mathbf{A} \vdash \mathbf{A}}$$

False axiom

$$\frac{}{\vdash \{\ \mathit{false}\ \}\ \ s_1\ \ \{\ \mathit{false}\ ,\ \mathit{false}\ \}}$$

Assumpt-intro-axiom

$$\frac{\mathcal{A} \vdash \mathbf{A}}{\mathbf{A_0}, \mathcal{A} \vdash \mathbf{A}}$$

Assumpt-elim-axiom

$$\frac{\begin{array}{c}\mathcal{A} \vdash \mathbf{A_0} \\ \mathbf{A_0}, \mathcal{A} \vdash \mathbf{A}\end{array}}{\mathcal{A} \vdash \mathbf{A}}$$

Strength

$$\frac{\begin{array}{c}P' \Rightarrow P \\ \mathcal{A} \vdash \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}\end{array}}{\mathcal{A} \vdash \{\ P'\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}}$$

Weak

$$\frac{\begin{array}{c}\mathcal{A} \vdash \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \} \\ Q_n \Rightarrow Q'_n \\ Q_e \Rightarrow Q'_e\end{array}}{\mathcal{A} \vdash \{\ P\ \}\ \ s_1\ \ \{\ Q'_n\ ,\ Q'_e\ \}}$$

Conjunction

$$\frac{\begin{array}{c}\mathcal{A} \vdash \{\ P'\ \}\ \ s_1\ \ \{\ Q'_n\ ,\ Q'_e\ \} \\ \mathcal{A} \vdash \{\ P''\ \}\ \ s_1\ \ \{\ Q''_n\ ,\ Q''_e\ \}\end{array}}{\mathcal{A} \vdash \{\ P' \wedge P''\ \}\ \ s_1\ \ \{\ Q'_n \wedge Q''_n\ ,\ Q'_e \wedge Q''_e\ \}}$$

Disjunction

$$\frac{\begin{array}{c}\mathcal{A} \vdash \{\ P'\ \}\ \ s_1\ \ \{\ Q'_n\ ,\ Q'_e\ \} \\ \mathcal{A} \vdash \{\ P''\ \}\ \ s_1\ \ \{\ Q''_n\ ,\ Q''_e\ \}\end{array}}{\mathcal{A} \vdash \{\ P' \vee P''\ \}\ \ s_1\ \ \{\ Q'_n \vee Q''_n\ ,\ Q'_e \vee Q''_e\ \}}$$

Invariant

$$\frac{\mathcal{A} \vdash \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}}{\mathcal{A} \vdash \{\ P \wedge W\ \}\ \ s_1\ \ \{\ Q_n \wedge W\ ,\ Q_e \wedge W\ \}}$$

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

Substitution

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

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

all-rule

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

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

ex-rule

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

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

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

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:

```
CilInstruction   ::= Instruction   // CIL instructions as described in Section 5.1
                   | TryCatchBlock
                   | TryFinallyBlock
TryCatchBlock    ::= .try {
                        list_of CilInstruction
                        Label : leave Label
                     }
                     catch CilType {
                        list_of CilInstruction
                        Label : leave Label
                     }
TryFinallyBlock  ::= .try {
                        list_of CilInstruction
                        Label : leave Label
                     }
                     .finally{
                        list_of CilInstruction
                        Label : endfinally
                     }
```

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:

$$\begin{array}{ll} \mathit{ExcTable} & := \mathit{List}[\mathit{ExcTableEntry}] \\ \mathit{ExcTableEntry} & := [\mathit{start} : \mathit{Label}, \mathit{end} : \mathit{Label}, \mathit{target} : \mathit{Label}, \mathit{excType} : \mathit{Type}] \end{array}$$

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 $[l_{start},\ l_{end}, l_{targ}, T_2]$ if and only if $l_{start} \leq l < l_{end}$ and $T_1 \preceq T_2$. Control is then transferred to l_{targ}.

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

$$
\begin{array}{l}
\mathit{table} : \ \mathit{CilProgram} \rightarrow \ \mathit{ExcTable} \\
\quad \mathit{table}\ [\,] \ = [\,] \\
\quad \mathit{table} \left(\begin{array}{l} \texttt{.try\{}\ w\ \texttt{\}} \\ \texttt{catch}\ T\ \{\ z\ \} \end{array} \right) \#xs \ \ l\ T = [l_n, l_{m+1}, l_i, T] + \mathit{table}\ w + \mathit{table}\ z + \mathit{table}\ xs \\[2ex]
\quad \mathit{table} \left(\begin{array}{l} \texttt{.try\{}\ w\ \texttt{\}} \\ \texttt{.finally}\ \{\ z\ \} \end{array} \right) \#xs \ \ l\ T \ = [l_n, l_{m+1}, l_i, \mathit{any}] + \mathit{table}\ w + \mathit{table}\ z + \mathit{table}\ xs \\[2ex]
\quad \mathit{table}\ x\#xs = \mathit{table}\ xs \\[2ex]
\mathit{where}\ w = l_n : \mathit{Inst}_n,\ ...,\ l_m : \mathit{Inst}_m \ \ \mathit{and} \ \ z = \ l_i : \mathit{Inst}_m,\ ...,\ l_j : \mathit{Inst}_j
\end{array}
$$

The function *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 *handler* is defined as follows:

$$
\begin{array}{lll}
\mathit{handler} : \mathit{ExcTable} \ \times \ \mathit{Label} \ \times \ \mathit{Type} \ \longrightarrow \ \mathit{Label} & & \\
\quad \mathit{handler}\ [\,]\ l\ T & = & \mathit{arbitrary} \\
\quad \mathit{handler}\ (x\#xs)\ l\ T & = & \textbf{if}\ (x.\mathit{start} \leq l < x.\mathit{end} \wedge \ T \preceq x.\mathit{excType})\ \textbf{then}\ (x.\mathit{target}) \\
 & & \textbf{else}\ (\mathit{handler}\ xs\ l\ T)
\end{array}
$$

Rules

In the following, we present the rules for `throw` instruction, and .try catch and .try .finally structures. Furthermore, we extent Bannwart and Müller's method call rule [6] to handle exceptions.

Throw. To prove the instruction specification { P } l: 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 *handler*. To simplify the notation, we write $\mathit{handler}(l)$ for the application of the function *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 `throw` is defined as follows:

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

Similar to the source logic, we use the variable $excV$ to store the current exception value. The instruction throw takes the top of the stack (an object of type *Exception*), and assigns it to $excV$. Then, control is transferred to the label where the exception is caught. In the implication $P \Rightarrow E_{handler(l)}\ [s(0)/excV]$, the replacement $[s(0)/excV]$ is used to assign the top of the stack to $excV$. Note that the instruction 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 .try block as well as the body of the catch block is a sequence of instruction specifications. The precondition of the .try instruction is defined by the precondition of the first instruction in the .try block. In a similar way, the precondition of the catch block is defined by the precondition of the first instruction of the catch block. To show that a .try catch block is valid, we need to show that all the instructions in the .try and the catch blocks are valid.

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

$$\frac{\forall\ InstSpec \in seq_1 + seq_2 : \mathcal{A} \vdash\ InstSpec}{\mathcal{A} \vdash\ \mathsf{.try}\{\ seq_1\ \}\ \ \mathsf{catch}\ \ T\ \{\ seq_2\ \}}$$

Try-finally. The semantics of the .try .finally structure is as follows. First, the instructions of the .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 .try block terminates with a 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 .finally block triggers an exception, control is transfer to the place where the exception is caught.

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

1. All instructions in the .try and .finally blocks are valid (except for the leave instructions in the .try block).
2. For all instruction specifications { P} l: leave k in the .try block, the following holds:
    - the precondition of the leave implies the precondition of the first instruction in the .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:

$$\begin{aligned} seq_1 &= \{P_1\}\ l_1 : I_1\ \ ...\ \ \{P_i\} l_i : \mathsf{leave}\ l_n \\ seq_2 &= \{P_j\}\ l_j : I_j\ \ ...\ \ \{P_k\} l_k : \mathsf{endfinally} \end{aligned}$$

Let seq'_1 denotes all the CIL instructions in seq_1 except for the leave instructions. The rule is defined as follows:

$$\frac{\begin{array}{c} \forall\ InstSpec \in seq'_1 + seq_2 : \mathcal{A} \vdash\ InstSpec \\ \\ for\ all\ \mathsf{leave}\ instructions \in seq_1\ of\ the\ form\ \{P_{i'}\} l_{i'} : \mathsf{leave}\ l_{n'}\ then \\ P_{i'} \Rightarrow\ P_j\ and \\ excV = null\ \wedge\ P_k \Rightarrow\ E_{l_{n'}} \\ \\ excV \neq null\ \wedge\ P_k \Rightarrow\ handler(l_i) \end{array}}{\mathcal{A} \vdash \mathsf{.try}\ \{\ seq_1\ \}\ \ \mathsf{.finally}\ \{\ seq_2\ \}}$$

The rule uses the sequence seq'_1 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 $\mathcal{A} \vdash \{\ P\ \}\ \ T{:}m\ \ \{\ Q_n\ ,\ Q_e\ \}$,

2. If $s(1)$ is not null, the precondition of the callvirt instruction, E_l, 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, E_l, 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, Q_n, implies the precondition of the successor instruction, E_{l+1}, with *Result* replaced by top of the stack, and

5. If $T{:}m$ triggers an exception ($excV$ 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 *handler* does not take the current exception table explicitly. The rule is defined as follows:

$$\dfrac{\begin{array}{c} \mathcal{A} \vdash \{\ P\ \}\quad T{:}m \quad \{\ Q_n\ ,\ Q_e\ \} \\[1em] s(1) \neq \mathit{null}\ \wedge\ E_l \Rightarrow\ P[s(1)/\mathit{this}, s(0)/p][\mathit{shift}(w)/Z] \\ s(1) = \mathit{null}\ \wedge\ E_l \Rightarrow\ E_{\mathit{handler}(l, \mathit{NullReference})} \\[1em] excV = \mathit{null}\ \wedge\ Q_n[s(0)/\mathit{result}][w/Z] \Rightarrow E_{l+1} \\ excV \neq \mathit{null}\ \wedge\ Q_e[w/z]\ \Rightarrow\ E_{\mathit{handler}(l,\ \tau(excV))} \end{array}}{\mathcal{A} \vdash \{E_l\}\ l : \mathsf{callvirt}\ T{:}m} \text{ cil invocation rule}$$

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 *Result*. The complete proof of example 5.1 is presented in Section 6.6. The bytecode specifications are the following:

$\{\ n > 1\ \}$	$IL005$: ldc 2
$\{\ n > 1\ \wedge\ s(0) = 2\ \}$	$IL006$: ldc 1
$\{\ n > 1\ \wedge\ s(0) = 1\ \wedge\ s(1) = 2\ \}$	$IL007$: stloc Result
$\{\ n > 1\ \wedge\ \mathit{Result} = 1\ \wedge\ s(0) = 2\ \}$	$IL008$: stloc i
$\{\ n > 1\ \wedge\ \mathit{Result} = 1\ \wedge\ i = 2\ \}$	$IL009$: br IL020

To be able to show that the bytecode specifications at labels $IL005 - IL008$ are valid, we need to show:

$$\textit{for all } l : \{IL005, ..., IL008\} : \quad \frac{E_l \Rightarrow wp(I_l)}{\vdash \{E_l\}\ l : I_l}$$

We prove it applying the *wp* definition. In the following, we present this proof.

- Case $IL005 : \dfrac{(n > 1) \Rightarrow wp(\textsf{ldc}\ 2)}{\vdash \{\ n > 1\ \}\ IL005 :\ \textsf{ldc}\ 2}$

  Applying the *wp* definition for ldc, we need to prove:

  $$(n > 1) \Rightarrow \textit{unshift}(\ (n > 1\ \wedge\ s(0) = 2)[2/s(0)]\ )$$

  We prove this implication as follows:

  $$\begin{array}{l} (n > 1) \Rightarrow \textit{unshift}(\ (n > 1\ \wedge\ s(0) = 2)[2/s(0)]\ ) \\ \quad [\textit{definition of replacement}] \\ (n > 1) \Rightarrow \textit{unshift}(n > 1\ \wedge\ 2 = 2) \\ \quad [\textit{definition of unshift}] \\ (n > 1) \Rightarrow (n > 1\ \wedge\ 2 = 2) \end{array}$$

  $\square$

- Case $IL006 : \dfrac{(n > 1\ \wedge\ s(0) = 2) \Rightarrow wp(\textsf{ldc}\ 1)}{\vdash \{\ n > 1\ \wedge\ s(0) = 2\ \}\ IL006 :\ \textsf{ldc}\ 1}$

  By the definition of *wp* for ldc, we need to prove:

  $$(n > 1\ \wedge\ s(0) = 2) \Rightarrow \textit{unshift}(\ (n > 1\ \wedge\ s(0) = 1\ \wedge\ s(1) = 2)[1/s(0)]\ )$$

  We show that this implication holds applying the definitions of replacement and *unshift*:

  $$\begin{array}{l} (n > 1\ \wedge\ s(0) = 2) \Rightarrow \textit{unshift}(\ (n > 1\ \wedge\ s(0) = 1\ \wedge\ s(1) = 2)[1/s(0)]\ ) \\ \quad [\textit{definition of replacement}] \\ (n > 1\ \wedge\ s(0) = 2) \Rightarrow \textit{unshift}(n > 1\ \wedge\ 1 = 1\ \wedge\ s(1) = 2) \\ \quad [\textit{definition of unshift}] \\ (n > 1\ \wedge\ s(0) = 2) \Rightarrow (n > 1\ \wedge\ 1 = 1\ \wedge\ s(0) = 2) \end{array}$$

  $\square$

- Case $IL007: \dfrac{(n > 1 \ \wedge \ s(0) = 1 \ \wedge \ s(1) = 2) \Rightarrow wp(\mathsf{stloc}\ Result)}{\vdash \{ \ n > 1 \ \wedge \ s(0) = 1 \ \wedge \ s(1) = 2 \ \} \ \ IL007 : \ \mathsf{stloc}\ Result}$

  By the definition of *wp* for stloc, we need to show:

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

  We prove this implication in the following way:

  $$\begin{array}{l} (n > 1 \wedge s(0) = 1 \wedge s(1) = 2) \Rightarrow shift(n > 1 \wedge Result = 1 \wedge s(0) = 2)[s(0)/Result] \\ \quad [definition \ \ of \ \ shift] \\ (n > 1 \wedge s(0) = 1 \wedge s(1) = 2) \Rightarrow (n > 1 \wedge Result = 1 \wedge s(1) = 2)[s(0)/Result] \\ \quad [definition \ \ of \ \ replacement] \\ (n > 1 \wedge s(0) = 1 \wedge \ s(1) = 2) \Rightarrow (n > 1 \wedge s(0) = 1 \wedge s(1) = 2) \end{array}$$

  □

- Case $IL008: \dfrac{(n > 1 \ \wedge \ Result = 1 \ \wedge \ s(0) = 2) \Rightarrow wp(\mathsf{stloc}\ i)}{\vdash \{ \ n > 1 \ \wedge \ Result = 1 \ \wedge \ s(0) = 2 \ \} \ \ IL008 : \ \mathsf{stloc}\ i}.$

  By the definition of *wp* for stloc, we need to show:

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

  We prove it as follows:

  $$\begin{array}{l} (n > 1 \ \wedge \ Result = 1 \ \wedge \ s(0) = 2) \Rightarrow shift(n > 1 \ \wedge \ Result = 1 \ \wedge \ i = 2) \ [s(0)/i] \\ \quad [definition \ \ of \ \ shift] \\ (n > 1 \ \wedge \ Result = 1 \ \wedge \ s(0) = 2) \Rightarrow (n > 1 \ \wedge \ Result = 1 \ \wedge \ i = 2) \ [s(0)/i] \\ \quad [definition \ \ of \ \ replacement] \\ (n > 1 \ \wedge \ Result = 1 \ \wedge \ s(0) = 2) \Rightarrow (n > 1 \ \wedge \ Result = 1 \ \wedge \ s(0) = 2) \end{array}$$

  □

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 ∇_P to translate the source proof, one function, ∇_B to translate a proof tree, one function ∇_S for instructions, and one for expressions ∇_E. The function ∇_P yields a list of CIL proofs (a derivation in the bytecode logic); and the function ∇_B yields a CIL proof. The functions ∇_E and ∇_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:

$$\textbf{datatype}\ \textit{Proof}\ = \textbf{list\_of}\ \ \textit{ProofTree}$$

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 *routine implementation rule*, the *class rule*, and the language independent rules. The leaves of the CIL proof trees are bytecode proofs (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:

$$\begin{array}{lll}\textbf{datatype}\ \mathit{CILProof} & = \textbf{list\_of} & \mathit{CILProofTree} \\ \textbf{datatype}\ \mathit{BytecodeProof} & = \textbf{list\_of} & \mathit{InstrSpec} \\ \textbf{datatype}\ \mathit{InstrSpec} & = \mathit{Precondition\ Label\ Instruction} & \end{array}$$

Translating Source Proofs. The main translation function ∇_P takes a source proof, and yields a CIL proof. This function is defined using the function ∇_B, which translates the source proof tree. To be able to translate the source proof, we use the function ∇_S for the translation of instructions, and the function ∇_E for expressions. The signatures of these functions are defined as follows:

$$\begin{array}{ll}\nabla_P & : \mathit{Proof} \rightarrow \mathit{CILProof} \\ \nabla_B & : \mathit{ProofTree} \rightarrow \mathit{CILProofTree} \\ \nabla_E & : \mathit{Precondition} \times \mathit{Expression} \times \mathit{Postcondition} \times \mathit{Label} \rightarrow \mathit{BytecodeProof} \\ \nabla_S & : \mathit{ProofTree} \times \mathit{Label} \times \mathit{Label} \times \mathit{Label} \rightarrow \mathit{BytecodeProof}\end{array}$$

The definitions of the function ∇_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 ∇_P is defined as follows:

$$\nabla_P\ (p) \ = \nabla_B(tree_1) + ... + \nabla_B(tree_n)$$

The definition of the function ∇_B is presented in the following section. The functions ∇_E and ∇_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 rule 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 ∇_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 ∇_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:

$$T_{imp} \equiv \frac{\mathit{Tree}_1}{\mathcal{A} \vdash \{\ \tau(\mathit{Current}) = T\ \wedge\ P\ \}\ \ \mathit{imp}(T, m)\ \ \{\ Q_n\ ,\ Q_e\ \}}$$

$$T_{tm} \equiv \frac{\mathit{Tree}_2}{\mathcal{A} \vdash \{\ \tau(\mathit{Current}) \prec T\ \wedge\ P\ \}\qquad T{:}m\qquad \{\ Q_n\ ,\ Q_e\ \}}$$

where Tree_1 and Tree_2 are the derivations used to prove the Hoare triples of $\mathit{imp}(T, m)$ and $T{:}m$ respectively.

The definition of the translation is the following:

$$\nabla_B \left(\frac{T_{imp} \qquad T_{tm}}{\mathcal{A} \vdash \Big\{\ \tau(\mathit{Current}) \preceq T\ \wedge\ P\ \Big\}\qquad T{:}m\qquad \Big\{\ Q_n\ ,\ Q_e\ \Big\}} \right) =$$

$$\frac{\nabla_B(\ T_{imp}\ ) \qquad \nabla_B(\ T_{tm}\ )}{\mathcal{A} \vdash \Big\{\ \tau(\mathit{Current}) \preceq T\ \wedge\ P\ \Big\}\qquad T{:}m\qquad \Big\{\ Q_n\ ,\ Q_e\ \Big\}}\ \text{cil class rule}$$

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 ∇_B to T_{imp} and T_{tm}.

6.2.2 Subtype Rule

Let T_{sm} be the following proof tree:

$$T_{sm} \equiv \frac{Tree_1}{\mathcal{A} \vdash \{\ P\ \}\ \ S{:}m\ \ \{\ Q_n\ ,\ Q_e\ \}}$$

The definition of the translation of the subtype rule is the following:

$$\nabla_B \left(\frac{S \preceq T \qquad T_{sm}}{\mathcal{A} \vdash \left\{\ \tau(Current) \preceq S\ \wedge\ P\ \right\}\ \ T{:}m\ \ \left\{\ Q_n\ ,\ Q_e\ \right\}} \right) =$$

$$\frac{S \preceq T \qquad \nabla_B(\ T_{sm}\ )}{\mathcal{A} \vdash \left\{\ \tau(Current) \preceq S\ \wedge\ P\ \right\}\ \ T{:}m\ \ \left\{\ Q_n\ ,\ Q_e\ \right\}}\ \text{cil subtype rule}$$

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:

$$T_{body} \equiv \frac{Tree_1}{\mathcal{A}, \{P\}\ \ T@m\ \ \{Q_n,\ Q_e\} \vdash \{\ P\ \}\ \ body(T@m)\ \ \{\ Q_n\ ,\ Q_e\ \}}$$

The translation is defined as follows:

$$\nabla_B \left(\frac{T_{body}}{\mathcal{A} \vdash \left\{\ P\ \right\}\ \ T@m\ \ \left\{\ Q_n\ ,\ Q_e\ \right\}} \right) =$$

$$\cfrac{\cfrac{\nabla_S(T_{body})}{\mathcal{A}, \{P\} T@m \{Q_n, Q_e\} \vdash \left\{\ P\ \right\}\ \ body\_cil(T@m)\ \ \left\{\ Q_n\ ,\ Q_e\ \right\}}\ \text{cil body rule}}{\mathcal{A} \vdash \left\{\ P\ \right\}\ \ T@m\ \ \left\{\ Q_n\ ,\ Q_e\ \right\}}\ \text{cil r. impl. rule}$$

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 ∇_S. The function ∇_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:

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

False axiom

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

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

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

Assumpt-intro-axiom

Let T_A be the following proof tree:

$$T_A \equiv \frac{\mathit{Tree}_1}{\mathcal{A} \vdash \mathbf{A}}$$

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:

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

Assumpt-elim-axiom

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

$$T_A \equiv \frac{\mathit{Tree}_1}{\mathcal{A} \vdash \mathbf{A_0}} \qquad T_{A_0} \equiv \frac{\mathit{Tree}_2}{\mathbf{A_0}, \mathcal{A} \vdash \mathbf{A}}$$

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:

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

Strength

Let T_{S_1} be the following proof tree:

$$T_{S_1} \equiv \frac{\mathit{Tree}_1}{\mathcal{A} \vdash \{\ P\ \} \quad s_1 \quad \{\ Q_n\ ,\ Q_e\ \}}$$

The definition is:

$$\nabla_B \left(\frac{P' \Rightarrow P \qquad T_{S_1}}{\mathcal{A} \vdash \{\ P'\ \} \quad s_1 \quad \{\ Q_n\ ,\ Q_e\ \}} \right) =$$

$$\frac{P' \Rightarrow P \qquad \nabla_B\,(\ T_{S_1})}{\mathcal{A} \vdash \{\ P'\ \} \quad s_1 \quad \{\ Q_n\ ,\ Q_e\ \}} \text{ cil strength rule}$$

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.

Weak

Let T_{S_1} be the following proof tree:

$$T_{S_1} \equiv \frac{Tree_1}{\mathcal{A} \vdash \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}}$$

The translation is defined as following:

$$\nabla_B \left(\frac{T_{S_1} \qquad Q_n \Rightarrow Q'_n \qquad Q_e \Rightarrow Q'_e}{\mathcal{A} \vdash \{\ P\ \}\ \ s_1\ \ \{\ Q'_n\ ,\ Q'_e\ \}} \right) =$$

$$\frac{\nabla_B\ (\ T_{S_1}) \qquad Q_n \Rightarrow Q'_n \qquad Q_e \Rightarrow Q'_e}{\mathcal{A} \vdash \{\ P\ \}\ \ s_1\ \ \{\ Q'_n\ ,\ Q'_e\ \}}\ \text{cil weak rule}$$

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:

$$T_{S_1} \equiv \frac{Tree_1}{\mathcal{A} \vdash \{\ P'\ \}\ \ s_1\ \ \{\ Q'_n\ ,\ Q'_e\ \}} \qquad T_{S_2} \equiv \frac{Tree_2}{\mathcal{A} \vdash \{\ P''\ \}\ \ s_1\ \ \{\ Q''_n\ ,\ Q''_e\ \}}$$

The translation of the conjunction rule yields the application of the CIL conjunction rule. The definition is:

$$\nabla_B \left(\frac{T_{S_1} \qquad T_{S_2}}{\mathcal{A} \vdash \{\ P' \wedge P''\ \}\ \ s_1\ \ \{\ Q'_n \wedge Q''_n\ ,\ Q'_e \wedge Q''_e\ \}} \right) =$$

$$\frac{\nabla_B\ (\ T_{S_1}) \qquad \nabla_B\ (\ T_{S_2})}{\mathcal{A} \vdash \{\ P' \wedge P''\ \}\ \ s_1\ \ \{\ Q'_n \wedge Q''_n\ ,\ Q'_e \wedge Q''_e\ \}}\ \text{cil conjunction rule}$$

Disjunction

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

$$T_{S_1} \equiv \frac{Tree_1}{\mathcal{A} \vdash \{\ P'\ \}\ \ s_1\ \ \{\ Q'_n\ ,\ Q'_e\ \}} \qquad T_{S_2} \equiv \frac{Tree_2}{\mathcal{A} \vdash \{\ P''\ \}\ \ s_1\ \ \{\ Q''_n\ ,\ Q''_e\ \}}$$

Similar to the conjunction rule, this translation yields the application of the CIL disjunction rule. The definition is:

$$\nabla_B \left(\frac{T_{S_1} \qquad T_{S_2}}{\mathcal{A} \vdash \{\ P' \vee P''\ \}\ \ s_1\ \ \{\ Q_n' \vee Q_n''\ ,\ Q_e' \vee Q_e''\ \}} \right) =$$

$$\frac{\nabla_B (\ T_{S_1}) \qquad \nabla_B (\ T_{S_2})}{\mathcal{A} \vdash \{\ P' \vee P''\ \}\ \ s_1\ \ \{\ Q_n' \vee Q_n''\ ,\ Q_e' \vee Q_e''\ \}} \text{ cil disjunction rule}$$

Invariant

Let T_{S_1} be the following proof tree:

$$T_{S_1} \equiv \frac{Tree_1}{\mathcal{A} \vdash \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}}$$

The translation is defined as follows:

$$\nabla_B \left(\frac{T_{S_1}}{\mathcal{A} \vdash \{\ P \wedge W\ \}\ \ s_1\ \ \{\ Q_n \wedge W\ ,\ Q_e \wedge W\ \}} \right) =$$

$$\frac{\nabla_B (\ T_{S_1})}{\mathcal{A} \vdash \{\ P \wedge W\ \}\ \ s_1\ \ \{\ Q_n \wedge W\ ,\ Q_e \wedge W\ \}} \text{ cil invariant rule}$$

Substitution

Let T_{S_1} be the following proof tree:

$$T_{S_1} \equiv \frac{Tree_1}{\mathcal{A} \vdash \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}}$$

The translation is the following:

$$\nabla_B \left(\frac{T_{S_1}}{\mathcal{A} \vdash \{\ P[t/Z]\ \}\ \ s_1\ \ \{\ Q_n[t/Z]\ ,\ Q_e[t/Z]\ \}} \right) =$$

$$\frac{\nabla_B (\ T_{S_1})}{\mathcal{A} \vdash \{\ P[t/Z]\ \}\ \ s_1\ \ \{\ Q_n[t/Z]\ ,\ Q_e[t/Z]\ \}} \text{ cil substitution rule}$$

All-rule

Let T_{S_1} be the following proof tree:

$$T_{S_1} \equiv \frac{\mathit{Tree}_1}{\mathcal{A} \vdash \{\ P[Y/Z]\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}}$$

The translation is:

$$\nabla_B \left(\frac{T_{S_1}}{\mathcal{A} \vdash \{\ P[Y/Z]\ \}\ \ s_1\ \ \{\ \forall Z : Q_n\ ,\ \forall Z : Q_e\ \}} \right) =$$

$$\frac{\nabla_B\ (\ T_{S_1})}{\mathcal{A} \vdash \{\ P[Y/Z]\ \}\ \ s_1\ \ \{\ \forall Z : Q_n\ ,\ \forall Z : Q_e\ \}}$$

Ex-rule

Let T_{S_1} be the following proof tree:

$$T_{S_1} \equiv \frac{\mathit{Tree}_1}{\mathcal{A} \vdash \{\ P\ \}\ \ s_1\ \ \{\ Q_n[Y/Z]\ ,\ Q_e[Y/Z]\ \}}$$

The translation is:

$$\nabla_B \left(\frac{T_{S_1}}{\mathcal{A} \vdash \{\ \exists Z : P\ \}\ \ s_1\ \ \{\ Q_n[Y/Z]\ ,\ Q_e[Y/Z]\ \}} \right) =$$

$$\frac{\nabla_B\ (\ T_{S_1})}{\mathcal{A} \vdash \{\ \exists Z : P\ \}\ \ s_1\ \ \{\ Q_n[Y/Z]\ ,\ Q_e[Y/Z]\ \}}$$

6.3 Proof Translation of Expressions

Expressions are translated using the function ∇_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:

$$\nabla_E\ \ : \mathit{Precondition} \times \mathit{Expression} \times \mathit{Postcondition} \times \mathit{Label} \rightarrow \mathit{BytecodeProof}$$

In ∇_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 stloc or ldloc instead of starg and ldarg when the variable is an argument.

6.3.1 Constants

Constants are translated using the ldc instruction. The constant is loaded on top of the stack. The precondition of the instruction specification is $Q \wedge \mathit{unshift}(P[c/s(0)])$. The translation is defined as follows:

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

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

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_{start} implies the weakest precondition of the instruction ldc using the postcondition $\mathit{shift}(Q) \wedge P$. This implication holds as follows:

$$
\begin{array}{l}
Q\ \wedge\ \mathit{unshift}(P[c/s(0)])\ \ \mathit{implies}\ \ \mathit{wp}(\mathsf{ldc}\ c, \mathit{shift}(Q) \wedge P) \\
\Leftrightarrow\quad [\mathit{definition\ of\ wp}] \\
Q\ \wedge\ \mathit{unshift}(P[c/s(0)])\ \ \mathit{implies}\ \ \mathit{unshift}(\ \mathit{shift}(Q) \wedge P[c/s(0)]\ ) \\
\Leftrightarrow\quad [\mathit{definition\ of\ unshift}] \\
Q\ \wedge\ \mathit{unshift}(P[c/s(0)])\ \ \mathit{implies}\ \ Q\ \wedge\ \mathit{unshift}(P[c/s(0)]) \\
\square
\end{array}
$$

6.3.2 Variables

Similar to the translation of constants, variables are translated loading the variable on top of the stack using the instruction ldloc. The definition is:

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

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

6.3.3 Binary Expressions

To translate binary expressions $e_1\ op\ e_2$, first the expression e_1 is translated applying the function ∇_E, then the expression e_2 is translated, and finally the binary operation is added. The translation is defined as follows:

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

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

The instruction bin_{op} 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 $\mathit{shift}(Q) \wedge P$. The definition of wp of bin_{op} is $(\mathit{shift}(E_{l+1}))[(s(1) ops(0))/s(1)]$, so we get:

$$\mathit{shift}^2(Q)\ \wedge\ \mathit{shift}(P[s(1)\ op\ s(0)/s(1)])$$

The precondition of the translation of e_2 is obtained replacing $s(0)$ by e_2, and then applying *unshift* (this result is the application of wp definition for ldloc). Finally, the precondition of e_1 is the obtained replacing $s(0)$ by e_1, and applying the function *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 ∇_E. The definition is:

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

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

6.4 Poof Translation of Instructions

Instructions are translated using the function ∇_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 ∇_S is defined as a composition of the translations of the proof's sub-trees. The signature is:

$$\nabla_S \quad : \mathit{ProofTree} \times \mathit{Label} \times \mathit{Label} \times \mathit{Label} \rightarrow \mathit{BytecodeProof}$$

In ∇_S, the three labels are: (1) *start* for the first label of the resulting bytecode; (2) *next* for the label after the resulting bytecode; this is for instance used in the translation of an `else` branch to determine where to jump at the end; (3) *exc* for the jump target when an exception is thrown. In this chapter, the label *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 7 with the translation of `rescue` clauses, and in Chapter 8 with the translation of `try-catch`, `try-finally`, and `throw` instructions.

6.4.1 Assignment Axiom

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

$$\nabla_S \left(\frac{}{\mathcal{A} \vdash \left\{ \begin{array}{l} (\mathit{safe}(e) \ \wedge \ P[e/x]) \vee \\ (\neg \mathit{safe}(e) \ \wedge \ Q_e) \end{array} \right\} \ x := e \ \left\{ P \ , \ Q_e \right\}} , l_{start}, l_{next}, l_{exc} \right) =$$

$$\nabla_E \left(\left(\begin{array}{l} (\mathit{safe}(e) \ \wedge \ P[e/x]) \vee \\ (\neg \mathit{safe}(e) \ \wedge \ Q_e) \end{array} \right) , \ e \ , \ (\mathit{shift}(\mathit{safe}(e) \ \wedge \ P[e/x]) \ \wedge \ s(0) = e) \ , \ l_{start} \right)$$

$$\left\{ \ \mathit{shift}(\mathit{safe}(e) \ \wedge \ P[e/x]) \wedge \ s(0) = e \ \right\} \quad l_b : \mathsf{stloc} \ x$$

The precondition of the application of the function ∇_E is the precondition of the assignment rule. The postcondition of application of the function ∇_E is

$$(\mathit{shift}(\mathit{safe}(e) \ \wedge \ P[e/x]) \ \wedge \ s(0) = e)$$

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 ∇_E. Also, the postcondition of ∇_E implies the precondition at the label l_b. Applying the definition of the weakest precondition of stloc x to P we can show that the instruction specification at l_b is valid:

$$\begin{array}{l} \mathit{shift}(\mathit{safe}(e) \ \wedge \ P[e/x]) \wedge \ s(0) = e \ \ \mathit{implies} \ \ \mathit{wp}(\mathsf{stloc}\ x, P) \\ \Leftrightarrow \quad [\mathit{definition\ of\ wp}] \\ \mathit{shift}(\mathit{safe}(e) \ \wedge \ P[e/x]) \wedge \ s(0) = e \ \ \mathit{implies} \ \ \mathit{shift}(P[s(0)/x]\ ) \\ \square \end{array}$$

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:

$$T_{S_1} \equiv \frac{\mathit{Tree}_1}{\{\ P\ \} \quad s_1 \quad \{\ Q_n\ ,\ R_e\ \}} \qquad T_{S_2} \equiv \frac{\mathit{Tree}_2}{\{\ Q_n\ \} \quad s_2 \quad \{\ R_n\ ,\ R_e\ \}}$$

The definition of the translation is the following:

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

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

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

6.4.3 Conditional Rule

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

$$T_{S_1} \equiv \frac{\mathit{Tree}_1}{\mathcal{A} \vdash \left\{ \; P \;\wedge\; e \; \right\} \;\; s_1 \;\; \left\{ \; Q_n \;,\; Q_e \; \right\}}$$

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

The translation is defined as follows:

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

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

In this translation, the expression of the conditional is translated using ∇_E. After the translation ∇_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 ∇_S. Otherwise, s_2 is translated and control is transferred to the next instruction. Here, the argument l_{next} is used to obtain the label of the next instruction. This argument avoids the generation of nop instructions at the end of the translation.

The translation of the expression e establishes $(\mathit{shift}(P) \;\wedge\; s(0) = e)$ 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 $e = \mathit{true}$. 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:

$$\nabla_S\left(\frac{}{\mathcal{A} \vdash \{\ P\ \}\ \ \texttt{check}\ e\ \texttt{end}\ \ \{\ (P\ \wedge\ e\ )\ ,\ (P\ \wedge\ \neg e\ )\ \}},\ l_{start}, l_{next}, l_{exc}\right) =$$

	$\nabla_E(\ P,\ e,\ \{shift(P)\ \wedge\ s(0) = e\},\ l_{start}\ )$
$\{shift(P)\ \wedge\ s(0) = e\}$	l_b : brtrue l_{next}
$\{P\ \wedge\ \neg e\}$	l_c : newobj $Exception()$
$\{shift(P)\ \wedge\ \neg e\ \wedge s(0) = new(\$, Exception)\}$	l_d : throw

The translation of the expression e establishes $shift(P) \wedge s(0) = e$. This postcondition implies the precondition at label l_b. If $s(0)$ evaluates to *true*, then e holds, and the precondition $shift(P)\ \wedge\ s(0) = e$ implies the normal postcondition of the check rule. If $s(0)$ evaluates to *false*, $shift(P)\ \wedge\ s(0) = e$ implies $P\ \wedge\ \neg e$, and the precondition at l_b implies the precondition at l_d. Finally, $shift(P)\ \wedge\ \neg e\ \wedge s(0) = new(\$, Exception)$ 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:

$$T_{S_1} \equiv \frac{Tree_1}{\mathcal{A} \vdash \{\ \neg e\ \wedge\ I\ \ \}\ \ s_1\ \ \{\ I\ ,\ R_e\ \}}$$

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 brfalse transfer control to the instruction at label l_b if the until expression is *false*. The definition is:

$$\nabla_S \left(\frac{T_{S_1}}{\mathcal{A} \vdash \left\{ \; I \; \right\} \quad \texttt{until } e \texttt{ loop } s_2 \texttt{ end} \quad \left\{ \; (I \;\wedge\; e) \;,\; R_e \; \right\}} , \; l_{start}, l_{next}, l_{exc} \right) =$$

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

The instruction br at label l_{start} preserves I, which is the precondition of the successor instruction. The translation of the expression e establishes $\mathit{shift}(I) \ \wedge\ s(0) = e$ 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 $\neg e \ \wedge\ I$ for the successor l_b (the first instruction of the translation of T_{S_1}) and $I \ \wedge\ e$ 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:

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

The translation is the following:

$$\nabla_S \left(\frac{}{\mathcal{A} \vdash \left\{ \; P' \; \right\} \quad x := y.S@a \quad \left\{ \; P \;,\; Q_e \; \right\}} , \; l_{start}, l_{next}, l_{exc} \right) =$$

$$\begin{array}{ll}
\{\ P'\ \} & l_{start} : \mathsf{ldloc}\ y \\
\{\ s(0) = y\ \wedge\ \mathit{shift}(P')\} & l_b : \mathsf{ldfld}\ S@a \\
\{\ s(0) = \$(iv(y, S@a))\ \wedge \mathit{shift}(P[\$(iv(y, S@a))/x])\} & l_c : \mathsf{stloc}\ x
\end{array}$$

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 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 $y \neq void$, and $P[\$(instvar(y, S@a))/x]$ holds.

6.4.7 Write Attribute Axiom

Let P' be the following precondition:

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

The definition of the translation is the following:

$$\nabla_S \left(\frac{}{\mathcal{A} \vdash \left\{ \ P' \ \right\} \ \ y.S@a := e \ \ \left\{ \ P \ , \ Q_e \ \right\}} , \ l_{start}, l_{next}, l_{exc} \right) =$$

$$\begin{array}{ll} \{ \ P' \ \} & l_{start} : \mathsf{ldloc} \ y \\ \multicolumn{2}{l}{\nabla_E(\{s(0) = y \ \wedge \ shift(P')\}, \ e, \{s(1) = y \ \wedge \ s(0) = e \ \wedge \ shift^2(P')\} \ , \ l_b)} \\ \{s(1) = y \ \wedge \ s(0) = e \ \wedge \ shift^2(P')\} & l_c : \mathsf{stfld} \ S@a \end{array}$$

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 stfld. If the target object y is void, then the instruction stfld triggers an exception.

The instruction ldloc y establishes $s(0) = y \ \wedge \ shift(P')$ because it pushes y on top of the stack. The translation of the expression e establishes

$$s(1) = y \ \wedge \ s(0) = e \ \wedge \ shift^2(P')$$

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:

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

The translation is defined as follows:

$$\nabla_S \left(\frac{\dfrac{\mathit{Tree}_1}{\mathcal{A} \vdash \left\{ \ P \ \right\} \ \ T : m(p) \ \ \left\{ \ Q_n \ , \ Q_e \ \right\}}}{\mathcal{A} \vdash \left\{ \ P' \ \right\} \ \ x = y.T : m(e) \ \ \left\{ \ Q_n[x/\mathit{result}] \ , \ Q_e \ \right\}}, \ l_{start}, l_{next}, l_{exc} \right) =$$

$$\begin{array}{ll}
\{P'\} & l_{start} : \mathsf{ldloc}\ y \\
\multicolumn{2}{l}{\nabla_E(\{\mathit{shift}(P') \ \wedge \ s(0) = y\}, \ e, \{\mathit{shift}^2(P') \ \wedge \ s(1) = y \ \wedge \ s(0) = e\} \ , \ l_b)} \\
\{\mathit{shift}^2(P') \ \wedge \ s(1) = y \ \wedge \ s(0) = e\} & l_c : \mathsf{callvirt}\ T : m \\
\{Q_n[s(0)/\mathit{result}]\} & l_d : \mathsf{stloc}\ x
\end{array}$$

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

$$\vdash \{\mathit{shift}^2(P') \ \wedge \ s(1) = y \ \wedge \ s(0) = e\} \ l_c : \mathsf{callvirt}\ T : m$$

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:

$$T_{tm} \equiv \frac{\mathit{Tree}_1}{\mathcal{A} \vdash \left\{ \ P \ \right\} \ \ T : m(p) \ \ \left\{ \ Q_n \ , \ Q_e \ \right\}}$$

The translation of the CIL method invocation is:

$$\nabla_B\,(\ T_{tm})$$

$$\begin{array}{l} s(1) \neq null \wedge\ shift^2(P')\ \wedge\ s(1) = y\ \wedge\ s(0) = e\ \ \Rightarrow\ \ P \\ s(1) = null\ \ \ \wedge\ shift^2(P')\ \wedge\ s(1) = y\ \wedge\ s(0) = e\ \ \Rightarrow\ \ E_{l_{exc}} \end{array}$$

$$\frac{\begin{array}{c} excV = null\ \ \wedge\ Q_n[s(0)/result] \Rightarrow Q_n[s(0)/result] \\ excV \neq null\ \wedge\ Q_e\ \Rightarrow\ E_{l_{exc}} \end{array}}{\mathcal{A} \vdash \{shift^2(P')\ \wedge\ s(1) = y\ \wedge\ s(0) = e\}\ l_c : \textsf{callvirt}\ T : m}\ \text{cil invocation rule}$$

The function ∇_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 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 callvirt implies the precondition at the label l_{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_{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:

$$T_{tm} \equiv \frac{Tree_1}{\mathcal{A} \vdash \{\ P\ \}\ \ x := y.T{:}m(e)\ \ \{\ Q_n\ ,\ Q_e\ \}}$$

In the translation:

$$\nabla_S \left(\frac{T_{tm}}{\mathcal{A} \vdash \{\ P[w/Z]\ \}\ \ x := y.T{:}m(e)\ \ \{\ Q_n[w/Z]\ ,\ Q_e[w/Z]\ \}},\ l_{start}, l_{next}, l_{exc} \right)$$

we first apply the translation function ∇_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, ..., v_n with the default values. To translate this rule, we first obtain the translation of the initialization using the translation function

∇_S. The variables are initialized using the assignment $v_i := \mathit{init}(T_i)$ for all variables v_1, ..., v_n. After that, the translation of the instruction s is added using ∇_S. The translation is defined as follows:

$$\nabla_S\left(\frac{\dfrac{\mathit{Tree}_1}{\mathcal{A}\vdash\left\{\begin{array}{l}P \wedge v_1 = \mathit{init}(T_1)\\ \wedge \ldots \wedge v_n = \mathit{init}(T_n)\end{array}\right\}\ s\ \left\{\ Q_n\ ,\ Q_e\ \right\}}}{\mathcal{A}\vdash\left\{\ P\ \right\}\ \texttt{local}\ \ T_1\ v_1;\ \ldots\ T_n\ v_n;\ s\ \left\{\ Q_n\ ,\ Q_e\ \right\}},\ l_{start}, l_{next}, l_{exc}\right) =$$

$$\nabla_S\left(\frac{\mathit{Tree}_1}{\mathcal{A}\vdash\left\{\ P\ \right\}\ v_1 := \mathit{init}(T_1)\ \left\{\ P\ \wedge\ v_1 = \mathit{init}(T_1)\ ,\ \mathit{false}\ \right\}},\ l_{start}, l_b, l_{exc}\right)$$

$$\nabla_S\left(\frac{\mathit{Tree}_1}{\mathcal{A}\vdash\left\{\begin{array}{l}P\ \wedge\\ v_1 = \mathit{init}(T_1)\end{array}\right\}\ v_2 := \mathit{init}(T_2)\ \left\{\left(\begin{array}{l}P\ \wedge\\ v_1 = \mathit{init}(T_1)\ \wedge\\ v_2 = \mathit{init}(T_2)\end{array}\right),\ \mathit{false}\right\}},\ l_b, l_c, l_{exc}\right)$$

...

$$\nabla_S\left(\frac{\mathit{Tree}_1}{\mathcal{A}\vdash\left\{\begin{array}{l}P\ \wedge\\ v_1 = \mathit{init}(T_1)\\ \wedge\ldots\wedge\\ v_{n-1} = \mathit{init}(T_{n-1})\end{array}\right\}\ v_n := \mathit{init}(T_n)\ \left\{\left(\begin{array}{l}P\ \wedge\\ v_1 = \mathit{init}(T_1)\\ \wedge\ldots\wedge\\ v_n = \mathit{init}(T_n)\end{array}\right),\ \mathit{false}\right\}},\ l_c, l_d, l_{exc}\right)$$

$$\nabla_S\left(\frac{\mathit{Tree}_1}{\mathcal{A}\vdash\left\{\ P\ \wedge\ v_1 = \mathit{init}(T_1)\ \wedge\ \ldots\wedge\ v_n = \mathit{init}(T_n)\ \right\}\ s\ \left\{\ Q_n\ ,\ Q_e\ \right\}},\ l_d, l_{next}, l_{exc}\right)$$

The bytecode for $v_1 := \mathit{init}(T_1)$ establishes $P\ \wedge\ v_1 = \mathit{init}(T_1)$, which is the precondition of the first instruction of the bytecode for $v_2 := \mathit{init}(T_2)$. Applying the same 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 *make* is invoked. To invoke the routine *make*, the target object x and the argument e are pushed on top of the stack. Then, the routine is invoked using the instruction callvirt. The proof of the 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:

$$\nabla_S \left(\begin{array}{l} \dfrac{\{\ P\ \}\quad T@make(p)\quad \{\ Q_n\ ,\ Q_e\ \}}{\mathcal{A} \vdash \left\{ P \left[\begin{array}{l} new(\$, T)/Current, \\ \$\langle T\rangle/\$, \\ e/p \end{array} \right] \right\}} , \ l_{start}, l_{next}, l_{exc} \\ \\ \quad x := \texttt{create}\ \{T\}.make(e) \\ \\ \quad \{\ Q_n[x/Current],\ Q_e\ \} \end{array} \right) =$$

$\{P[new(\$, T)/Current, \$\langle T\rangle/\$, e/p]\}$	l_{start} : newobj T
$\{(shift(P)[s(0)/Current, e/p])\}$	l_b : stloc x
$\{x \neq Void\ \wedge\ P[x/Current, e/p]\}$	l_c : ldloc x
∇_E ($\{x \neq Void\ \wedge\ shift(P[x/Current, e/p])\ \wedge\ s(0) = x\}, e,$	
$\{x \neq Void\ \wedge\ shift^2(P[x/Current, e/p])\ \wedge\ s(1) = x\ \wedge\ s(0) = e\}, l_d)$	
$\{x \neq Void\ \wedge\ shift^2(P[x/Current, e/p])\ \wedge\ s(1) = x\ \wedge\ s(0) = e\}$	l_e : callvirt $T@make$

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 ∇_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:

$$\nabla_S\left(\frac{\dfrac{Tree_1}{\mathcal{A}\vdash\{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}}\qquad P'\Rightarrow P}{\mathcal{A}\vdash\{\ P'\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}}\ ,\ l_{start}, l_{next}, l_{exc}\right) =$$

$$\begin{array}{ll}\{P'\} & l_{start} : \mathsf{nop} \\ & \nabla_S\left(\dfrac{Tree_1}{\mathcal{A}\vdash\left\{\ P\ \right\}\ \ s_1\ \ \left\{\ Q_n\ ,\ Q_b\ \right\}Q_e},\ l_b, l_{next}, l_{exc}\right)\end{array}$$

The instruction nop at label l_{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 ∇_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, and Q'_e are not modified by the translation. The translation is defined as follows:

$$\nabla_S\left(\frac{\dfrac{Tree_1}{\mathcal{A}\vdash\{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}}\qquad \begin{array}{c}Q_n\Rightarrow Q'_n\\ Q_e\Rightarrow Q'_e\end{array}}{\mathcal{A}\vdash\{\ P\ \}\ \ s_1\ \ \{\ Q'_n\ ,\ Q'_e\ \}},\ l_{start}, l_{next}, l_{exc}\right) =$$

$$\nabla_S \left(\frac{Tree_1}{\mathcal{A} \vdash \left\{ \; P \; \right\} \; s_1 \; \left\{ \; Q_n \; , \; Q_e \; \right\}}, \quad l_{start}, l_b, l_{exc} \right)$$
$\{Q_n\}$ $\quad l_b$: 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:

$$T_a \equiv \frac{Tree_1}{\mathcal{A} \vdash \{ \; P' \; \} \;\; s_1 \;\; \{ \; Q'_n \; , \; Q'_e \; \}} \qquad T_b \equiv \frac{Tree_2}{\mathcal{A} \vdash \{ \; P'' \; \} \;\; s_1 \;\; \{ \; Q''_n \; , \; Q''_e \; \}}$$

To translate the conjunction rule:

$$\nabla_S \left(\frac{T_a \qquad\qquad T_b}{\left\{ \; P' \; \wedge \; P'' \; \right\} \;\; s_1 \;\; \left\{ \; Q'_n \; \wedge \; Q''_n \; , \; Q'_e \; \wedge \; Q''_e \; \right\}}, \; l_{start}, l_{next}, l_{exc} \right) =$$

we first create the CIL proofs for the two hypotheses:

$$\nabla_S (\; T_a, \; l_{start}, l_{next}, l_{exc})$$

and

$$\nabla_S (\; T_b, \; l_{start}, l_{next}, l_{exc})$$

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 callvirt, their bytecode specification:

$$\{A_i\} \; l_i : \; instr_i$$

and

$$\{B_i\} \; l_i : \; instr_i$$

we obtain

$$\{A_i \ \wedge \ B_i\} \ l_i : instr_i$$

If the generated instruction is callvirt, the proof trees have the following form:

$$\dfrac{\dfrac{Tree_1}{\mathcal{A} \vdash \{\ P'\ \} \quad T{:}m \quad \{\ Q'_n\ ,\ Q'_e\ \}} \qquad A_i \Rightarrow s(1) \neq null \ \wedge \ P'[s(1)/this, s(0)/p][shift(w')/Z'] \qquad Q'_n[s(0)/result][w'/Z'] \Rightarrow E_{l_i+1}}{\mathcal{A} \vdash \{A_i\} \ l_i : \mathsf{callvirt} \ T{:}m} \ \text{cil invocation rule}$$

$$\dfrac{\dfrac{Tree_2}{\mathcal{A} \vdash \{\ P''\ \} \quad T{:}m \quad \{\ Q''_n\ ,\ Q''_e\ \}} \qquad B_i \Rightarrow s(1) \neq null \ \wedge \ P''[s(1)/this, s(0)/p][shift(w'')/Z''] \qquad Q''_n[s(0)/result][w''/Z''] \Rightarrow E_{l_i+1}}{\mathcal{A} \vdash \{B_i\} \ l_i : \mathsf{callvirt} \ T{:}m} \ \text{cil invocation rule}$$

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 $\mathcal{A} \vdash \{A_i \wedge \ B_i\} \ l_i : \mathsf{callvirt} \ T{:}m$ applying the CIL conjunction rule as follows:

$$\dfrac{\dfrac{\dfrac{Tree_1}{\mathcal{A} \vdash \{\ P'\ \} \quad T{:}m \quad \{\ Q'_n\ ,\ Q'_e\ \}} \qquad \dfrac{Tree_2}{\mathcal{A} \vdash \{\ P''\ \} \quad T{:}m \quad \{\ Q''_n\ ,\ Q''_e\ \}}}{\mathcal{A} \vdash \{\ P' \ \wedge \ P''\ \} \quad T{:}m \quad \{\ Q'_n \ \wedge \ Q''_n\ ,\ Q'_e \ \wedge \ Q''_e\ \}} \ \text{conj rule} \qquad (A_i \ \wedge \ B_i) \Rightarrow s(1) \neq null \ \wedge \ (P' \ \wedge \ P'')[s(1)/this, s(0)/p][shift(w')/Z'][shift(w'')/Z''] \qquad (Q'_n \ \wedge \ Q''_n)[s(0)/result][w'/Z'][w''/Z''] \Rightarrow E_{l_i+1}}{\mathcal{A} \vdash \{A_i \wedge \ B_i\} \ l_i : \mathsf{callvirt} \ T{:}m} \ \text{invok}$$

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:

$$\nabla_S\left(\frac{\dfrac{Tree_1}{\mathcal{A}\vdash\{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}}}{\mathcal{A}\vdash\{\ P\ \wedge\ W\ \}\ \ s_1\ \ \{\ Q_n\ \wedge\ W\ ,\ Q_e\ \wedge\ W\ \}},\ l_{start}, l_{next}, l_{exc}\right) =$$

$$\nabla_S\left(\frac{Tree_1}{\{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}},\ l_{start}, l_{next}, l_{exc}\right)$$

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:

$$\nabla_S\left(\frac{\dfrac{Tree_1}{\mathcal{A}\vdash\{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}}}{\mathcal{A}\vdash\{\ P[t/Z]\ \}\ \ s_1\ \ \{\ Q_n[t/Z]\ ,\ Q_e[t/Z]\ \}},\ l_{start}, l_{next}, l_{exc}\right) =$$

$$\nabla_S\left(\frac{Tree_1}{\{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}},\ l_{start}, l_{next}, l_{exc}\right)$$

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*:

$$\frac{\begin{array}{c}\{\ \tau(\mathit{Current}) \prec \mathit{MATH}\ \wedge\ n > 1\ \}\quad \mathit{MATH{:}sum}\quad \{\ \mathit{Result} = (n * (n+1))/2\ ,\ \mathit{false}\ \}\\ \{\ \tau(\mathit{Current}) = \mathit{MATH}\ \wedge\ n > 1\ \}\quad \mathit{impl(MATH,\ sum)}\quad \{\ \mathit{Result} = (n * (n+1))/2\ ,\ \mathit{false}\ \}\end{array}}{\{\ \tau(\mathit{Current}) \preceq \mathit{MATH}\ \wedge\ n > 1\ \}\quad \mathit{MATH{:}sum}\quad \{\ \mathit{Result} = (n * (n+1))/2\ ,\ \mathit{false}\ \}}$$

In the hypothesis, the proof for the virtual routine *MATH:sum* is obtained applying the translation function ∇_B to the hypothesis of the source rule. This application returns the following application of the *strength rule*:

$$\frac{\{\ \mathit{false}\ \}\quad \mathit{MATH{:}sum}\quad \{\ \mathit{Result} = (n * (n+1))/2\ ,\ \mathit{false}\ \}}{\{\ \tau(\mathit{Current}) \prec \mathit{MATH}\ \wedge\ n > 1\ \}\quad \mathit{MATH{:}sum}\quad \{\ \mathit{Result} = (n * (n+1))/2\ ,\ \mathit{false}\ \}}$$

The proof for the second hypothesis is obtained by the translation function ∇_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 ∇_S. This proof is presented in Figure 6.1.

$$\frac{\dfrac{\mathit{proof\ \ Figure}\ 6.1}{\{\ \tau(\mathit{Current}) = \mathit{MATH}\ \wedge\ n > 1\ \}\quad \mathit{body\_cil(MATH@sum)}\quad \{\ \mathit{Result} = (n * (n+1))/2\ ,\ \mathit{false}\ \}}}{\{\ \tau(\mathit{Current}) = \mathit{MATH}\ \wedge\ n > 1\ \}\quad \mathit{MATH@sum}\quad \{\ \mathit{Result} = (n * (n+1))/2\ ,\ \mathit{false}\ \}}$$

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 *IL*001-*IL*004. Lines *IL*005-*IL*008 are produced by the translation of the body of the `from` part of the loop. The *weak* and *strength* rules are translated in lines *IL*009−*IL*011. Lines *IL*012−*IL*019 translate the *body* of the loop. Finally, the *until expression* is translated in lines *IL*020 − *IL*025.

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 *IL*009 − *IL*010, and lines *IL*011 − *IL*012 (the translation of the *weak* and *strength* rule). These two implications are proven since the proofs have been developed for the source

$\{ \ n > 1 \ \}$	$IL001$: ldc 0
$\{ \ n > 1 \ \wedge \ s(0) = 0 \ \}$	$IL002$: stloc Result
$\{ \ n > 1 \ \wedge \ \mathit{Result} = 0 \ \}$	$IL003$: ldc 0
$\{ \ n > 1 \ \wedge \ \mathit{Result} = 0 \ \wedge \ s(0) = 0 \ \}$	$IL004$: stloc i
// from body	
$\{ \ n > 1 \ \wedge \ \mathit{Result} = 0 \ \wedge \ i = 0 \ \}$	$IL005$: ldc 1
$\{ \ n > 1 \ \wedge \ \mathit{Result} = 0 \ \wedge \ i = 0 \ \wedge \ s(0) = 1 \ \}$	$IL006$: stloc Result
$\{ \ n > 1 \ \wedge \ \mathit{Result} = 1 \ \wedge \ i = 0 \ \}$	$IL007$: ldc 2
$\{ \ n > 1 \ \wedge \ \mathit{Result} = 1 \ \wedge \ s(0) = 2 \ \}$	$IL008$: stloc i
// weak and strength rules	
$\{ \ n > 1 \ \wedge \ \mathit{Result} = 1 \ \wedge \ i = 2 \ \}$	$IL009$: nop
$\{ \ \mathit{Result} = ((i-1)*i)/2) \ \wedge \ n+1 \geq i \ \wedge \ i > 1 \ \}$	$IL010$: br IL020
$\{ \ i \neq n+1 \ \wedge \ \mathit{Result} = ((i-1)*i)/2) \ \wedge \ n+1 \geq i \ \wedge \ i > 1 \ \}$	$IL011$: nop
// loop body	
$\left\{ \begin{array}{l} i \neq n+1 \ \wedge \ \mathit{Result} + i = (i*(i+1))/2 \ \wedge \\ n+1 \geq i \ \wedge \ i > 1 \end{array} \right\}$	$IL012$: ldloc Result
$\left\{ \begin{array}{l} i \neq n+1 \ \wedge \ \mathit{Result} + i = (i*(i+1))/2 \ \wedge \\ n+1 \geq i \ \wedge \ i > 1 \ \wedge \ s(0) = \mathit{Result} \end{array} \right\}$	$IL013$: ldloc i
$\left\{ \begin{array}{l} i \neq n+1 \ \wedge \ \mathit{Result} + i = (i*(i+1))/2 \ \wedge \\ n+1 \geq i \ \wedge \ i > 1 \ \wedge \ s(1) = \mathit{Result} \ \wedge \ s(0) = i \end{array} \right\}$	$IL014$: add
$\left\{ \begin{array}{l} i \neq n+1 \ \wedge \ \mathit{Result} + i = (i*(i+1))/2 \ \wedge \\ n+1 \geq i \ \wedge \ i > 1 \ \wedge \ s(0) = \mathit{Result} + i \end{array} \right\}$	$IL015$: stloc Result
$\left\{ \begin{array}{l} i \neq n+1 \ \wedge \ \mathit{Result} = (i*(i+1))/2 \ \wedge \\ n+1 \geq i \ \wedge \ i > 1 \end{array} \right\}$	$IL016$: ldloc i
$\left\{ \begin{array}{l} i \neq n+1 \ \wedge \ \mathit{Result} = (i*(i+1))/2 \ \wedge \\ n+1 \geq i \ \wedge \ i > 1 \ \wedge \ s(0) = i \end{array} \right\}$	$IL017$: ldc 1
$\left\{ \begin{array}{l} i \neq n+1 \ \wedge \ \mathit{Result} = (i*(i+1))/2 \ \wedge \\ n+1 \geq i \ \wedge \ i > 1 \ \wedge \ s(1) = i \ \wedge \ s(0) = 1 \end{array} \right\}$	$IL018$: add
$\left\{ \begin{array}{l} i \neq n+1 \ \wedge \ \mathit{Result} = (i*(i+1))/2 \ \wedge \\ n+1 \geq i \ \wedge \ i > 1 \ \wedge \ s(0) = i+1 \end{array} \right\}$	$IL019$: stloc i
// until expression	
$\{ \ \mathit{Result} = ((i-1)*i)/2 \ \wedge \ n+1 \geq i \ \wedge \ i > 1 \ \}$	$IL020$: ldloc i
$\left\{ \begin{array}{l} \mathit{Result} = ((i-1)*i)/2 \ \wedge \ n+1 \geq i \ \wedge \ i > 1 \ \wedge \\ s(0) = i \end{array} \right\}$	$IL021$: ldloc n
$\left\{ \begin{array}{l} \mathit{Result} = ((i-1)*i)/2 \ \wedge \ n+1 \geq i \ \wedge \ i > 1 \ \wedge \\ s(1) = i \ \wedge \ s(0) = n \end{array} \right\}$	$IL022$: ldc 1
$\left\{ \begin{array}{l} \mathit{Result} = ((i-1)*i)/2 \ \wedge \ n+1 \geq i \ \wedge \ i > 1 \ \wedge \\ s(2) = i \ \wedge \ s(1) = n \ \wedge \ s(0) = 1 \end{array} \right\}$	$IL023$: add
$\left\{ \begin{array}{l} \mathit{Result} = ((i-1)*i)/2 \ \wedge \ n+1 \geq i \ \wedge \ i > 1 \ \wedge \\ s(1) = i \ \wedge \ s(0) = n+1 \end{array} \right\}$	$IL024$: ceq
$\left\{ \begin{array}{l} \mathit{Result} = ((i-1)*i)/2 \ \wedge \ n+1 \geq i \ \wedge \ i > 1 \ \wedge \\ s(0) = (i = n+1) \end{array} \right\}$	$IL025$: brfalse 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)**.**

$$\frac{Tree_1}{\mathcal{A} \vdash \left\{ \ P \ \right\} \ \ m \ \ \left\{ \ Q_n \ , \ Q_e \ \right\}} \qquad \textit{then}$$

$$\vdash \ \nabla_B \left(\frac{Tree_1}{\mathcal{A} \vdash \left\{ \ P \ \right\} \ \ m \ \ \left\{ \ Q_n \ , \ Q_e \ \right\}} \right)$$

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 $I_{l_{start}} ... I_{l_{end}}$, and their respective preconditions $E_{l_{start}} ... E_{l_{end}}$, and (3) the normal postcondition in the source logic implies the next precondition of the last generated instruction (if the last generated instruction is the last instruction of the method, we use the normal postcondition in the source logic), and (4) the exceptional postcondition in the source logic implies the precondition at the target label l_{exc} but considering the value stored in the stack of the bytecode, then every bytecode specification holds ($\vdash \{E_l\}\ I_l$). The theorem is the following:

Theorem 4 (Soundness of Instruction Translator)**.**

$$\begin{array}{l}
\vdash \dfrac{\mathit{Tree}_1}{\mathcal{A} \vdash \left\{\ P\ \right\}\ \ s_1\ \ \left\{\ Q_n\ ,\ Q_e\ \right\}} \quad \wedge \\
(I_{l_{start}} ... I_{l_{end}}) = \nabla_S \left(\dfrac{\mathit{Tree}_1}{\mathcal{A} \vdash \left\{\ P\ \right\}\ \ s_1\ \ \left\{\ Q_n\ ,\ Q_e\ \right\}},\ l_{start}, l_{end+1}, l_{exc} \right) \quad \wedge \\
\big(Q_n \quad \Rightarrow \quad E_{l_{end+1}}\big) \quad \wedge \\
(\ (Q_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \quad \Rightarrow \quad E_{l_{exc}}) \quad \wedge \\
\Rightarrow \\
\forall\ l\ \in\ l_{start}\ ...\ l_{end} : \vdash \{E_l\}\ I_l
\end{array}$$

The 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.

```
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 $\neg$, $\wedge$, $\vee$, *AndThen*, *OrElse*, *Xor*, *implies*, expressions equality $<$, $>$, $\leq$, $\geq$, and the type functions *ConformsTo*, *IsEqual*, and *IsNotEqual*. Expressions are constants, local variables and arguments, attributes, 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.

```
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
```

$$\begin{array}{lll}
\textbf{datatype}\ \mathit{TypeExpr} & = & \textbf{EType}\ \ \mathit{EiffelType} \\
 & | & \textbf{Type}\ \ \mathit{Expr}
\end{array}$$

$$\begin{array}{lll}
\textbf{datatype}\ \mathit{Expr} & = & \textbf{ConstInt}\ \ \mathit{Int} \\
 & | & \textbf{RefVar}\ \ \mathit{VarId} \\
 & | & \textbf{Att}\ \ \mathit{ObjID}\ \mathit{Attrib} \\
 & | & \textbf{CallR}\ \ \mathit{CallRoutine} \\
 & | & \textbf{Precursor}\ \ \mathit{EiffelType}\ \mathit{Routine}\ \mathit{Argument} \\
 & | & \textbf{Create}\ \ \mathit{EiffelType}\ \mathit{Routine}\ \mathit{Argument} \\
 & | & \textbf{Plus}\ \mathit{Expr}\ \mathit{Expr} \\
 & | & \textbf{Minus}\ \mathit{Expr}\ \mathit{Expr} \\
 & | & \textbf{Mul}\ \mathit{Expr}\ \mathit{Expr} \\
 & | & \textbf{Div}\ \mathit{Expr}\ \mathit{Expr} \\
 & | & \textbf{Old}\ \ \mathit{Expr} \\
 & | & \textbf{Bool}\ \ \mathit{BoolExpr} \\
 & | & \textbf{Void}
\end{array}$$

$$\begin{array}{lll}
\textbf{datatype}\ \mathit{CallRoutine} & = & \textbf{Call}\ \ \mathit{Expr}\ \mathit{Routine}\ \mathit{Argument}
\end{array}$$

$$\begin{array}{lll}
\textbf{datatype}\ \mathit{Argument} & = & \textbf{Argument}\ \ \mathit{Expr}
\end{array}$$

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

$$\begin{array}{lll}
\textbf{datatype}\ \mathit{EiffelType} & = & \textbf{Boolean} \\
 & | & \textbf{Integer} \\
 & | & \textbf{EClass}\ (\mathit{cID} : \mathit{ClassID}) \\
 & | & \textbf{None}
\end{array}$$

Variables, attributes and routines are defined as follows:

$$\begin{array}{lll}
\textbf{datatype}\ \mathit{Var} & = & \textbf{Var}\ \ \mathit{VarID}\ \mathit{EiffelType} \\
 & | & \textbf{Result}\ \ \mathit{EiffelType} \\
 & | & \textbf{Current}\ \ \mathit{EiffelType} \\
\textbf{datatype}\ \mathit{Attrib} & = & \textbf{Attr}\ \ (\mathit{aID} : \mathit{attribID})\ \mathit{EiffelType} \\
\textbf{datatype}\ \mathit{Routine} & = & \textbf{Routine}\ \ \mathit{RoutineID}\ \mathit{EiffelType}\ \mathit{EiffelType}
\end{array}$$

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.

$$\begin{aligned}\textbf{datatype}\ \mathit{CilType} &= \textbf{CilBoolean}\\ &\mid \textbf{CilInteger}\\ &\mid \textbf{Interface}\ \mathit{classID}\\ &\mid \textbf{CilClass}\ \mathit{classID}\\ &\mid \textbf{NullT}\end{aligned}$$

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

$$\begin{array}{lll@{\qquad}lll} \multicolumn{3}{l}{\nabla_{\mathit{Interface}} : \mathit{EiffelType} \to \mathit{CilType}} & \multicolumn{3}{l}{\nabla_{\mathit{Class}} : \mathit{EiffelType} \to \mathit{CilType}}\\ & \nabla_{\mathit{Interface}}(\textbf{Boolean}) & = \textbf{CilBoolean} & & \nabla_{\mathit{Class}}(\textbf{Boolean}) & = \textbf{CilBoolean}\\ & \nabla_{\mathit{Interface}}(\textbf{Integer}) & = \textbf{CilInteger} & & \nabla_{\mathit{Class}}(\textbf{Integer}) & = \textbf{CilInteger}\\ & \nabla_{\mathit{Interface}}(\textbf{EClass}\ n) & = \textbf{Interface}\ n & & \nabla_{\mathit{Class}}(\textbf{EClass}\ n) & = \textbf{CilClass}\ n\\ & \nabla_{\mathit{Interface}}(\textbf{None}) & = \textbf{NullT} & & \nabla_{\mathit{Class}}(\textbf{None}) & = \textbf{NullT} \end{array}$$

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

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

$$\begin{array}{l} \nabla_R : \mathit{Routine} \to \mathit{CilMethod}\\ \quad \nabla_R(\textbf{Routine}\ n\ t1\ t2) = (\textbf{Method}\ \ n\ \ (\nabla_{\mathit{Interface}}\ t1)\ \ (\nabla_{\mathit{Interface}}\ t2)) \end{array}$$

7.1.4 Translation Functions

The translation of the specification relies on five translation functions: (1) ∇_C takes a boolean expression and returns a function that takes two stores and a state and returns a value; (2) ∇_{Exp} translates expressions; (3) ∇_T translates type functions (conforms to, is equal, and is not equal); (4) ∇_{Call} translates a routine call; and (5) ∇_{Arg} translates arguments. These functions use two object stores, the second one is used to evaluate old expressions. *State* is a mapping from variables to values ($\mathit{VarId} \to \mathit{Value}$). The signatures of these functions are the following:

$$\begin{array}{lll} \nabla_C : & \mathit{BoolExpr} & \to (\mathit{ObjectStore} \to \mathit{ObjectStore} \to \mathit{State} \to \mathit{Value})\\ \nabla_{\mathit{Exp}} : & \mathit{Expr} & \to (\mathit{ObjectStore} \to \mathit{ObjectStore} \to \mathit{State} \to \mathit{Value})\\ \nabla_T : & \mathit{TypeFunc} & \to (\mathit{ObjectStore} \to \mathit{ObjectStore} \to \mathit{State} \to \mathit{Value})\\ \nabla_{\mathit{Call}} : & \mathit{CallRoutine} & \to (\mathit{ObjectStore} \to \mathit{ObjectStore} \to \mathit{State} \to \mathit{Value})\\ \nabla_{\mathit{Arg}} : & \mathit{Argument} & \to (\mathit{ObjectStore} \to \mathit{ObjectStore} \to \mathit{State} \to \mathit{Value}) \end{array}$$

To clarify the translation of contracts, we repeat the definition of the data type *Value*:

$$
\begin{array}{lll}
\textbf{data type } \mathit{Value} = & \textbf{boolV } (\mathit{aB} : \mathit{Bool}) \\
& \mid \textbf{intV } (\mathit{aI} : \mathit{Int}) \\
& \mid \textbf{objV } \mathit{ClassId}\ \mathit{ObjId} \\
& \mid \textbf{voidV}
\end{array}
$$

where the notation $(aB : \mathit{Bool})$ means, given a value v of type boolean, $aB(v)$ returns its boolean expression. The definition of the function ∇_C is the following:

$$
\begin{array}{ll}
\nabla_C(\textbf{Const}\ b) & = \lambda\ (h_1, h_2 : \mathit{ObjectStore})\ (s : \mathit{State}) :\ (\mathit{boolV}\ \ b) \\
\nabla_C(\textbf{Neg}\ b) & = \lambda\ (h_1, h_2 : \mathit{ObjectStore})\ (s : \mathit{State}) : \\
& \quad (\mathit{boolV}\ \neg(aB(\nabla_C\ b\ h_1\ h_2\ s))) \\
\nabla_C(\textbf{And}\ b_1\ b_2) & = \lambda(h_1, h_2 : \mathit{ObjectStore})\ (s : \mathit{State}) : \\
& \quad (\mathit{boolV}(aB(\nabla_C\ b_1\ h_1\ h_2\ s)) \wedge (aB(\nabla_C\ b_2\ h_1\ h_2\ s))) \\
\nabla_C(\textbf{Or}\ b_1\ b_2) & = \lambda\ (h_1, h_2 : \mathit{ObjectStore})\ (s : \mathit{State}) : \\
& \quad (\mathit{boolV}\ \ (aB(\nabla_C\ b_1\ h_1\ h_2\ s)) \vee (aB(\nabla_C\ b_2\ h_1\ h_2\ s))) \\
\nabla_C(\textbf{AndThen}\ b_1\ b_2) & = \lambda(h_1, h_2 : \mathit{ObjectStore})\ (s : \mathit{State}) : \\
& \quad (\mathit{boolV}(aB(\nabla_C\ b_1\ h_1\ h_2\ s)) \wedge (aB(\nabla_C\ b_2\ h_1\ h_2\ s))) \\
\nabla_C(\textbf{OrElse}\ b_1\ b_2) & = \lambda(h_1, h_2 : \mathit{ObjectStore})\ (s : \mathit{State}) : \\
& \quad (\mathit{boolV}(aB(\nabla_C\ b_1\ h_1\ h_2\ s)) \vee (aB(\nabla_C\ b_2\ h_1\ h_2\ s))) \\
\nabla_C(\textbf{Xor}\ b_1\ b_2) & = \lambda(h_1, h_2 : \mathit{ObjectStore})\ (s : \mathit{State}) : \\
& \quad (\mathit{boolV}(aB(\nabla_C\ b_1\ h_1\ h_2\ s)) \vee (aB(\nabla_C\ b_2\ h_1\ h_2\ s))) \\
\nabla_C(\textbf{Impl}\ b_1\ b_2) & = \lambda(h_1, h_2 : \mathit{ObjectStore})\ (s : \mathit{State}) : \\
& \quad (\mathit{boolV}(aB(\nabla_C\ b_1\ h_1\ h_2\ s)) \longrightarrow (aB(\nabla_C\ b_2\ h_1\ h_2\ s))) \\
\nabla_C(\textbf{Eq}\ e_1\ e_2) & = \lambda\ (h_1, h_2 : \mathit{ObjectStore})\ (s : \mathit{State}) : \\
& \quad (\mathit{boolV}\ \ (aI(\nabla_{\mathit{Exp}}\ e_1\ h_1\ h_2\ s)) = (aI(\nabla_{\mathit{Exp}}\ e_2\ h_1\ h_2\ s))) \\
\nabla_C(\textbf{NotEq}\ e_1\ e_2) & = \lambda(h_1, h_2 : \mathit{ObjectStore})\ (s : \mathit{State}) : \\
& \quad (\mathit{boolV}(aI(\nabla_{\mathit{Exp}}\ e_1\ h_1\ h_2\ s)) \neq (aI(\nabla_{\mathit{Exp}}\ e_2\ h_1\ h_2\ s))) \\
\nabla_C(\textbf{Less}\ e_1\ e_2) & = \lambda(h_1, h_2 : \mathit{ObjectStore})\ (s : \mathit{State}) : \\
& \quad (\mathit{boolV}(aI(\nabla_{\mathit{Exp}}\ e_1\ h_1\ h_2\ s)) < (aI(\nabla_{\mathit{Exp}}\ e_2\ h_1\ h_2\ s))) \\
\nabla_C(\textbf{Greater}\ e_1\ e_2) & = \lambda(h_1, h_2 : \mathit{ObjectStore})\ (s : \mathit{State}) : \\
& \quad (\mathit{boolV}(aI(\nabla_{\mathit{Exp}}\ e_1\ h_1\ h_2\ s)) > (aI(\nabla_{\mathit{Exp}}\ e_2\ h_1\ h_2\ s)))
\end{array}
$$

$$\begin{aligned}
\nabla_C(\mathbf{LessE}\ e_1\ e_2) &= \lambda(h_1, h_2 : \mathit{ObjectStore})\ (s : \mathit{State}) : \\
&\qquad (\mathit{boolV}(\mathit{aI}(\nabla_{\mathit{Exp}}\ e_1\ h_1\ h_2\ s)) \leq (\mathit{aI}(\nabla_{\mathit{Exp}}\ e_2\ h_1\ h_2\ s))) \\
\nabla_C(\mathbf{GreaterE}\ e_1\ e_2) &= \lambda(h_1, h_2 : \mathit{ObjectStore})\ (s : \mathit{State}) : \\
&\qquad (\mathit{boolV}(\mathit{aI}(\nabla_{\mathit{Exp}}\ e_1\ h_1\ h_2\ s)) \geq (\mathit{aI}(\nabla_{\mathit{Exp}}\ e_2\ h_1\ h_2\ s))) \\
\nabla_C(\mathbf{Type}\ e) &= \lambda\ (h_1, h_2 : \mathit{ObjectStore})\ (s : \mathit{State}) :\ (\nabla_T\ e\ h_1\ h_2\ s))
\end{aligned}$$

The function ∇_T maps the Eiffel types to CIL. The Eiffel function *ConformsTo* is mapped to the function $\preceq_c$ (subtyping in CIL). Its types are translated to interfaces using the function $\nabla_{\mathit{Interface}}$. The function *IsEqual* is translated using the function $=$ (types equality in CIL), and the function *IsNotEqual* to $\neq$

Its types are translated to CIL classes using the function ∇_{Class}. The function ∇_t is defined as follows:

$$\begin{aligned}
\nabla_T(\mathbf{ConformsTo}\ t_1\ t_2) &= \lambda\ (h_1, h_2 : \mathit{ObjectStore})(s : \mathit{State}) : \\
&\qquad (\mathit{boolV}(\nabla_{\mathit{Interface}}(\nabla_{\mathit{Type}}\ t_1)) \preceq_c (\nabla_{\mathit{Interface}}(\nabla_{\mathit{Type}}\ t_2))) \\
\nabla_T(\mathbf{IsEqual}\ t_1\ t_2) &= \lambda\ (h_1, h_2 : \mathit{ObjectStore})(s : \mathit{State}) : \\
&\qquad (\mathit{boolV}(\nabla_{\mathit{Class}}(\nabla_{\mathit{Type}}\ t_1)) = (\nabla_{\mathit{Class}}(\nabla_{\mathit{Type}}\ t_2))) \\
\nabla_T(\mathbf{IsNotEqual}\ t_1\ t_2) &= \lambda\ (h_1, h_2 : \mathit{ObjectStore})(s : \mathit{State}) : \\
&\qquad (\mathit{boolV}(\nabla_{\mathit{Class}}(\nabla_{\mathit{Type}}\ t_1)) \neq (\nabla_{\mathit{Class}}(\nabla_{\mathit{Type}}\ t_2)))
\end{aligned}$$

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

$$\begin{aligned}
&\nabla_{\mathit{Type}} :\ \mathit{TypeExp} \to\ \mathit{EiffelType} \\
&\quad \nabla_{\mathit{Type}}(\mathbf{EType}\ t) \quad = t \\
&\quad \nabla_{\mathit{Type}}(\mathbf{Expression}\ e) \ = (\mathit{typeOf}\ \ e)
\end{aligned}$$

The function ∇_{Exp} translates local variables using the state s. Creation instructions are translated using the functions of the object store *new* and $\_(\_)$. The translation of old expressions uses the second *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):

$$\begin{aligned}
\_(\_) &: \mathit{ObjectStore}\ \times\ \mathit{Location} \to\ \mathit{Value} \\
\mathit{alive} &: \mathit{Value} \times \mathit{ObjectStore} \to \mathit{Bool} \\
\mathit{new} &: \mathit{ObjectStore}\ \times\ \mathit{ClassId} \to\ \mathit{Value} \\
\_\langle\_ := \_\rangle &: \mathit{ObjectStore}\ \times\ \mathit{Location}\ \times\ \mathit{Value} \to \mathit{ObjectStore} \\
\_\langle\_\rangle &: \mathit{ObjectStore}\ \times\ \mathit{ClassId} \to\ \mathit{ObjectStore}
\end{aligned}$$

The definition of ∇_{Exp} is:

$\nabla_{Exp}(\textbf{ConstInt}\ i) = \lambda\ (h_1, h_2 : ObjectStore)(s : State) : (intV\ i)$

$\nabla_{Exp}(\textbf{RefVar}\ v) = \lambda\ (h_1, h_2 : ObjectStore)(s : State) : (s(v))$

$\nabla_{Exp}(\textbf{Att}\ ob\ a) = \lambda\ (h_1, h_2 : ObjectStore)(s : State) : h_1\ (\ (Loc\ (aID\ a)\ ob)\ )$

$\nabla_{Exp}(\textbf{CallR}\ crt) = \lambda\ (h_1, h_2 : ObjectStore)(s : State) : (\nabla_{Call}\ crt\ h_1\ h2\ s)$

$\nabla_{Exp}(\textbf{Precursor}\ t_1\ rt\ par) = \lambda(h_1, h_2 : ObjectStore)(s : State) : (invokeValCIL\ h_1\ (\nabla_r\ rt)\ (s\ (Currentt_1))\ (\nabla_{arg}\ par\ h_1\ h_2\ s))$

$\nabla_{Exp}(\textbf{Create}\ t\ rt\ p) = \lambda\ (h_1, h_2 : ObjectStore)(s : State) : (\ new\ (h_1\langle(cID\ t)\rangle)\ (cID\ t)\ )$

$\nabla_{Exp}(\textbf{Plus}\ e_1\ e_2) = \lambda(h_1, h_2 : ObjectStore)(s : State) : (intV((aI(\nabla_{Exp}\ e_1\ h_1\ h_2\ s)) + (aI(\nabla_{Exp}\ e_2\ h_1\ h_2\ s))))$

$\nabla_{Exp}(\textbf{Minus}\ e_1\ e_2) = \lambda(h_1, h_2 : ObjectStore)(s : State) : (intV((aI(\nabla_{Exp}\ e_1\ h_1\ h_2\ s)) - (aI(\nabla_{Exp}\ e_2\ h_1\ h_2\ s))))$

$\nabla_{Exp}(\textbf{Mul}\ e_1\ e_2) = \lambda(h_1, h_2 : ObjectStore)(s : State) : (intV((aI(\nabla_{Exp}\ e_1\ h_1\ h_2\ s)) * (aI(\nabla_{Exp}\ e_2\ h_1\ h_2\ s))))$

$\nabla_{Exp}(\textbf{Div}\ e_1\ e_2) = \lambda(h_1, h_2 : ObjectStore)(s : State) : (intV((aI(\nabla_{Exp}\ e_1\ h_1\ h_2\ s))div(aI(\nabla_{Exp}\ e_2\ h_1\ h_2\ s))))$

$\nabla_{Exp}(\textbf{Old}\ e) = \lambda\ (h_1, h_2 : ObjectStore)(s : State) : (\nabla_{Exp}\ e\ h_2\ h_2\ s)$

$\nabla_{Exp}(\textbf{Bool}\ b) = \lambda\ (h_1, h_2 : ObjectStore)(s : State) : (\nabla_C\ b\ h_1\ h_2\ s)$

$\nabla_{Exp}(\textbf{Void}) = \lambda\ (h_1, h_2 : ObjectStore)(s : State) : (VoidV)$

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

$\nabla_{Call}(\textbf{Call}\ e_1\ rt\ p) = \lambda\ (h_1, h_2 : ObjectStore)(s : State) : (\textbf{CilInvokeVal}\ h_1\ (\nabla_R\ rt)\ (\nabla_{Exp}\ e_1\ h_1\ h_2\ s)(\nabla_{Arg}\ p\ h_1\ h_2\ s))$

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

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

$\nabla_{Arg}(\textbf{Argument}\ e) = \lambda\ (h_1, h_2 : ObjectStore)(s : State) : (\nabla_{Exp}\ e\ h_1\ h_2\ s)$

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 ∇_C which produces the contracts in FOL. Following, we present the embedding of the contracts of the function *merge* presented in the introduction of this chapter. Its precondition is embedded as follows:

$$\text{Type (} \mathbf{ConformsTo} \text{ (Type (} \mathbf{RefVar}\ \mathit{other}\text{)) (EType } \mathit{LINKED\_LIST}[G]\text{))}$$
$$\text{Type (} \mathbf{IsEqual} \text{ (Type (} \mathbf{RefVar}\ \mathit{Current}\text{)) (Type (} \mathbf{RefVar}\ \mathit{other}\text{)))}$$

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

$$\text{Type (} \mathbf{IsEqual} \text{ (Type (} \mathbf{RefVar}\ \mathit{Current}\text{)) (EType } \mathit{LINKED\_LIST}[G]\text{))}$$

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

$$\lambda\ (h_1, h_2 : \mathit{ObjectStore})(s : \mathit{State}) :$$
$$\mathit{boolV}(\mathit{typeOf}\ \ \mathit{other})\ \preceq_c (\textbf{interface}\ \mathit{LINKED\_LIST}[G])$$
$$\lambda\ (h_1, h_2 : \mathit{ObjectStore})(s : \mathit{State}) :\ \mathit{boolV}(\mathit{typeOf}\ \ \mathit{Current})\ = (\mathit{typeOf}\ \ \mathit{other})$$

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

$$\lambda\ (h_1, h_2 : \mathit{ObjectStore})(s : \mathit{State}) :$$
$$\mathit{boolV}(\mathit{typeOf}\ \ \mathit{Current})\ = (\textbf{CilClass}\ \mathit{LINKED\_LIST}[G])$$

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

7.2 Proof Translator

In Section6.1 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 ∇_S and ∇_E to handle the Eiffel expressions `and then` and `or else`, and `rescue` clauses and once routines. These functions yield a sequence of Bytecode instructions and their specifications.

7.2.1 Transformation Function Basics

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

$$\begin{aligned}
\nabla_E \quad &: \mathit{Precondition} \times \mathit{Expression} \times \mathit{Postcondition} \times \mathit{Label} \rightarrow \mathit{BytecodeProof} \\
\nabla_S \quad &: \mathit{ProofTree} \times \mathit{Label} \times \mathit{Label} \times \mathit{Label} \rightarrow \mathit{BytecodeProof}
\end{aligned}$$

In ∇_E the label is used as the starting label of the translation. ProofTree is a derivation in the source logic. In ∇_S, the three labels are: (1) *start* for the first label of the resulting bytecode; (2) *next* for the label after the resulting bytecode; this is for instance used in the translation of an `else` branch to determine where to jump at the end; (3) *exc* for the jump target when an exception is thrown. The *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 ∇_E is presented in Section 7.2.2. The function ∇_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 **and then** e_2 is as follows: if the evaluation of e_1 yields *false*, then the evaluation of e_1 **and then** e_2 yields *false* without evaluating e_2; otherwise the evaluation of e_1 **and then** e_2 yields the result of the evaluation of e_2. The semantics of e_1 **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_e, and the constant *false* is pushed on top of the stack because e_1 **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_c), and control is transfer to the next instruction. Thus, the evaluation e_1 **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 $e_1 \ \wedge \ e_2$ is evaluated left to right; if e_1 evaluates to *false*, then e_2 is not evaluated. The translation is defined as follows:

$$\nabla_E(Q \wedge \mathit{unshift}(P[e_1 \;\wedge\; e_2/s(0)],\quad e_1 \; \textsf{and then} \; e_2 \;\; , \mathit{shift}(Q) \wedge P, l_a, l_{end}) =$$

$$\begin{array}{l}
\nabla_E(Q \wedge \mathit{unshift}(P[e_1 \;\wedge\; e_2/s(0)],\quad e_1,\quad \mathit{shift}(Q) \wedge P[s(0) \wedge e_2/s(0)], l_a, l_{end}) \\
\{\mathit{shift}(Q) \wedge P[s(0) \wedge e_2/s(0)]\} \; l_b : \; \textsf{brfalse} \; l_e \\
\nabla_E(Q \wedge \mathit{unshift}(P[(\mathit{true} \wedge e_2)/s(0)],\quad e_2,\quad \mathit{shift}(Q) \wedge P[(\mathit{true} \wedge s(0)/s(0)], l_c, l_{end}) \\
\{\mathit{shift}(Q) \wedge P[\mathit{true} \wedge s(0)/s(0)]\} \; l_d : \textsf{br} \; l_{end} \\
\{Q \wedge \mathit{unshift}(P[\mathit{false} \wedge e_2/s(0)])\} \; l_e : \textsf{ldc} \; \mathit{false}
\end{array}$$

The translation for e_1 establishes $\mathit{shift}(Q) \wedge P[s(0) \wedge e_2/s(0)]$ which is the precondition of the instruction at l_b. The instruction 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

$$Q \wedge \mathit{unshift}(P[(\mathit{true} \wedge e_2)/s(0)]$$

holds. If the top of the stack is *false*, we can prove

$$\mathit{shift}(Q) \wedge P[\mathit{true} \wedge s(0)/s(0)]$$

applying the definition of *wp*. Finally, the translation for e_2 establishes

$$\mathit{shift}(Q) \wedge P[(\mathit{true} \wedge s(0)/s(0)]$$

which is the precondition of its successor. Therefore, the translation is valid.

Expressions or else

The expressions **or else** are translated in a similar way than **and then** expressions. At label l_a, the expression e_1 is translated using the function ∇_E. This translation pushes the evaluation of e_1 on top of the stack. If e_1 is *true*, control is transferred to l_e 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 **and then** expressions, we assume the expression $e_1 \;\vee\; e_2$ is evaluated left to right, if e_1 evaluates to *true* then e_2 is not evaluated. The definition is:

$$\nabla_E(Q \wedge \mathit{unshift}(P[e_1 \,\textsf{or else}\, e_2/s(0)], e_1 \,\textsf{or else}\, e_2, \mathit{shift}(Q) \wedge P, l_a, l_{end}) =$$

$$\begin{array}{l}
\nabla_E(Q \wedge \mathit{unshift}(P[e_1 \vee \; e_2/s(0)], \quad e_1 \quad, \mathit{shift}(Q) \wedge P[s(0) \vee e_2/s(0)], l_a, l_{end}) \\
\{\mathit{shift}(Q) \wedge P[s(0) \vee e_2/s(0)]\} \; l_b : \; \textsf{brtrue} \; l_e \\
\nabla_E(Q \wedge \mathit{unshift}(P[(\mathit{false} \vee e_2)/s(0)], \quad e_2 \quad, \mathit{shift}(Q) \wedge P[(\mathit{false} \vee s(0))/s(0)], l_c, l_{end}) \\
\{\mathit{shift}(Q) \wedge P[\mathit{false} \vee s(0)/s(0)]\} \; l_d : \; \textsf{br} \; l_{end} \\
\{Q \wedge \mathit{unshift}(P[\mathit{true} \vee e_2/s(0)])\} \; l_e : \; \textsf{ldc} \; \mathit{true}
\end{array}$$

The translation for e_1 establishes $\mathit{shift}(Q) \wedge P[s(0) \vee e_2/s(0)]$ 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 $\mathit{shift}(Q) \wedge P[(\mathit{false} \vee s(0))/s(0)]$ which is the precondition of its successor l_d. Therefore, the translation for e_1 or else e_2 is valid.

7.2.3 Proof Translation of Instructions

In this section, we extend the translation function ∇_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:

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

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

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

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

The bytecode for 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 s_2 establishes $Retry \Rightarrow I_r \wedge \neg Retry \Rightarrow R_e$, 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 $T@m\_done$, $T@m\_exc$, and $T@m\_result$. Let P be the following precondition, where T_M_RES is a logical variable:

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

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

$$Q'_n \equiv \left\{ \begin{array}{l} T@m\_done\ \wedge\ \neg T@m\_exc\ \wedge \\ (Q_n \vee (\ P''\ \wedge\ Result = T\_M\_RES\ \wedge\ T@m\_result = T\_M\_RES\ )) \end{array} \right\}$$

$$Q'_e \equiv \left\{\ T@m\_done\ \wedge\ T@m\_exc\ \wedge\ (Q_e\ \vee P''')\ \right\}$$

Let T_{body} be the following proof tree:

$$T_{body} \equiv \cfrac{\mathit{Tree}_1}{\begin{array}{l}\mathcal{A}, \{P\} \;\; T@m \; \{Q'_n ,\; Q'_e\} \vdash \\[2ex] \left\{ \; P'[\mathit{false}/T@m\_done] \wedge \; T@m\_done \; \right\} \;\; \mathit{body}(T@m) \;\; \left\{ \; Q_n \; , \; Q_e \; \right\}\end{array}}$$

In this translation, the variable $T@m\_done$ is evaluated at label l_{start} . If its value is *true*, control is transferred at the end of the function (label l_k). If $T@m\_done$ evaluates to *false*, this variable is set to *true*, and the body of the function is translated using the translation function ∇_S. This translation assumes that the variable $T@m\_result$ is updated whenever the body of the function assigns to *Result*. If the body of the function triggers an exception, the exception is caught by the catch block at labels $l_i - l_l$. The translation is defined as follows:

$$\nabla_S \left(\cfrac{T_{body}}{\mathcal{A} \vdash \left\{ \; P \; \right\} \;\; T@m \;\; \left\{ \; Q'_n \; , \; Q'_e \; \right\}} , \; l_{start}, l_{next}, l_{exc} \right) =$$

```
$\{\ P\ \}$                                                            $l_a$ : ldsfld $T@m\_done$
$\{\ shift(P)\ \wedge\ s(0) = T@m\_done\}$                               $l_b$ : brtrue $l_m$
$\{\ P'\ \wedge\ \neg T@m\_done\}$                                        $l_c$ : ldc $true$
$\{\ shift(P')\ \wedge\ \neg T@m\_done\ \wedge\ s(0) = true\}$            $l_d$ : stsfld $T@m\_done$
$.try$
{
  $\nabla_S\ (T_{body}, l_e, l_f, l_i\ )$
  $\{Q_n\}$                                                            $l_f$ : ldloc $result$
  $\{shift(Q_n)\ \wedge\ s(0) = result\}$                                $l_g$ : stsfld $T@m\_result$
  $\{Q_n\ \wedge\ result = T@m\_result\}$                                $l_h$ : leave $l_m$
}
$catch\ \ System.Exception$
{
  $\{shift(Q_e)\ \wedge\ excV \neq null\ \ \wedge\ s(0) = excV\}$         $l_i$ : stloc $last\_exc$
  $\{Q_e\ \wedge\ last\_exc \neq null\ \}$                                $l_j$ : ldc $true$
  $\{shift(Q_e)\ \wedge\ last\_exc \neq null\ \wedge\ s(0) = true\}$      $l_k$ : stsfld $T@m\_exc$
  $\{Q_e\ \wedge\ last\_exc \neq null\ \wedge\ T@m\_exc = true\ \}$       $l_l$ : rethrow
}
$\{Q'\}$                                                               $l_m$ : ldsfld $T@m\_exc$
$\{shift(Q')\ \wedge\ s(0) = T@m\_exc\}$                                 $l_n$ : brfalse $l_q$
$\{T@m\_done\ \wedge\ T@m\_exc = true\ \wedge\ P'''\ \}$                   $l_o$ : ldsfld $last\_exc$
$\left\{\begin{array}{l} T@m\_done\ \wedge\ T@m\_exc = true\ \wedge\ P'''\ \wedge \\ s(0) = last\_exc \end{array}\right\}$      $l_p$ : throw
$\left\{\begin{array}{l} T@m\_done\ \wedge\ T@m\_exc = false\ \wedge \\ \left( Q_n\ \vee \left( \begin{array}{l} P''\ \wedge\ result = T@m\_RESULT\ \wedge \\ T@m\_result = T@m\_RESULT \end{array} \right) \right) \end{array}\right\}$      $l_q$ : ldsfld $T@m\_result$
$\left\{\begin{array}{l} T@m\_done\ \wedge\ T@m\_exc = false\ \wedge \\ \left( Q_n\ \vee \left( \begin{array}{l} P''\ \wedge\ result = T@m\_RESULT\ \wedge \\ T@m\_result = T@m\_RESULT \end{array} \right) \right)\ \wedge \\ s(0) = T@m\_result \end{array}\right\}$      $l_r$ : ret
```

$$\textit{where } Q' \equiv \left\{ \begin{array}{l} T@m\_done \wedge \\ \left(\begin{array}{l} (Q_n \vee \\ \left(\begin{array}{l} P'' \ \wedge \ T@m\_result = T@m\_RESULT \wedge \\ T@m\_exc = \mathit{false} \end{array} \right) \vee \\ (P''' \ \wedge \ T@m\_exc = \mathit{true}) \end{array} \right) \end{array} \right\}$$

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:

$$\frac{\begin{array}{c} \{\ \tau(\mathit{Current}) \prec \mathit{MATH} \ \wedge \ \mathit{true}\ \} \quad \mathit{MATH{:}safe\_division} \quad \{\ Q\ ,\ \mathit{false}\ \} \\ \{\ \tau(\mathit{Current}) = \mathit{MATH} \ \wedge \ \mathit{true}\ \} \quad \mathit{impl(MATH,\ safe\_division)} \quad \{\ Q\ ,\ \mathit{false}\ \} \end{array}}{\{\ \tau(\mathit{Current}) \preceq \mathit{MATH} \ \wedge \ \mathit{true}\ \} \quad \mathit{MATH{:}safe\_division} \quad \{\ Q\ ,\ \mathit{false}\ \}}$$

where

$$Q \equiv (y = 0 \Rightarrow \mathit{Result} = x) \ \wedge \ (y/ = 0 \rightarrow \mathit{Result} = x//y)$$

The triple $\{\ \tau(\mathit{Current}) \prec \mathit{MATH} \ \wedge \ \mathit{true}\ \}$ $\mathit{MATH{:}safe\_division}$ $\{\ Q\ ,\ \mathit{false}\ \}$ 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 $\mathit{impl}(\mathit{MATH}, \mathit{safe\_division}) = \mathit{MATH}@\mathit{safe\_division}$. The second hypothesis is obtained by the translation of the *routine implementation rule*. The proof in CIL is as follows:

$$\frac{\dfrac{\mathit{proof\ \ Figure}\ 7.1}{\{\ \mathit{true}\ \}\quad \mathit{body(MATH@safe\_division)} \quad \{\ Q\ ,\ \mathit{false}\ \}}}{\{\ \mathit{true}\ \}\quad \mathit{MATH@safe\_division} \quad \{\ Q\ ,\ \mathit{false}\ \}}$$

Figure 7.1 illustrates the translation of the body of the routine *safe_division*. This example is the result produced by the translation ∇_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 *IL*001 to *IL*007). Since the routine always terminates normally, the precondition of the instructions at labels *IL*017 and *IL*018 is *false*.

	.try {
$\{(y \neq 0 \wedge z = 0) \vee (y = 0 \wedge (z = 1 \vee z = 0))\}$	$IL001$: **ldloc** x
$\left\{\begin{array}{l}(y \neq 0 \wedge z = 0) \vee (y = 0 \wedge (z = 1 \vee z = 0)) \wedge \\ s(0) = x\end{array}\right\}$	$IL002$: **ldloc** y
$\left\{\begin{array}{l}(y \neq 0 \wedge z = 0) \vee (y = 0 \wedge (z = 1 \vee z = 0)) \wedge \\ s(1) = x \wedge\ s(0) = y\end{array}\right\}$	$IL003$: **ldloc** z
$\left\{\begin{array}{l}(y \neq 0 \wedge z = 0) \vee (y = 0 \wedge (z = 1 \vee z = 0)) \wedge \\ s(1) = x \wedge\ s(1) = y \wedge\ s(0) = z\end{array}\right\}$	$IL004$: **binop**$_{+}$
$\left\{\begin{array}{l}(y \neq 0 \wedge z = 0) \vee (y = 0 \wedge (z = 1 \vee z = 0)) \wedge \\ s(1) = x \wedge\ s(0) = y + z\end{array}\right\}$	$IL005$: **binop**$_{//}$
$\left\{\begin{array}{l}(y \neq 0 \wedge z = 0) \vee (y = 0 \wedge (z = 1 \vee z = 0)) \wedge \\ s(0) = x/(y + z)\end{array}\right\}$	$IL006$: **stloc** *Result*
$\left\{\begin{array}{l}(y = 0 \Rightarrow Result = x) \wedge \\ (y \neq 0 \Rightarrow Result = x/y)\end{array}\right\}$	$IL007$: **leave** $IL019$
	}
	catch System.Exception {
$\{y = 0\ \wedge\ z = 0 \wedge\ excV \neq null \wedge s(0) = excV\}$	$IL009$: **stloc** *last_exception*
$\{y = 0\ \wedge\ z = 0\}$	$IL010$: **ldc** 1
$\{y = 0\ \wedge\ z = 0 \wedge\ s(0) = 1\}$	$IL011$: **stloc** z
$\{y = 0\ \wedge\ z = 1\}$	$IL012$: **ldc** *true*
$\{y = 0\ \wedge\ z = 1 \wedge\ s(0) = true\}$	$IL013$: **stloc** *Retry*
$\{\neg Retry \Rightarrow false\ \wedge Retry \Rightarrow (y = 0 \vee z = 1)\}$	$IL014$: **ldloc** *Retry*
$\left\{\begin{array}{l}\neg Retry \Rightarrow false \wedge \\ Retry \Rightarrow (y = 0 \vee z = 1) \wedge\ s(0) = Retry\end{array}\right\}$	$IL015$: **brfalse** $IL017$
$\{y = 0\ \vee z = 1\}$	$IL016$: **br** 01
$\{false\}$	$IL017$: **ldloc** *last_exception*
$\{false\ \wedge\ s(0) = last\_exception\}$	$IL018$: **rethrow**
	}
$\left\{\begin{array}{l}(y = 0 \Rightarrow Result = x) \wedge \\ (y \neq 0 \Rightarrow Result = x/y)\end{array}\right\}$	$IL019$: **ldloc** *Result*
$\left\{\begin{array}{l}(y = 0 \Rightarrow Result = x) \wedge \\ (y \neq 0 \Rightarrow Result = x/y)\ \wedge\ s(0) = Result\end{array}\right\}$	$IL020$: **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 value_C, value_T, $\mathit{value}_{\mathit{Exp}}$, $\mathit{value}_{\mathit{Call}}$ and $\mathit{value}_{\mathit{Arg}}$ evaluate boolean expressions, Eiffel types, expressions, routine calls and arguments respectively. The theorem is the following:

Theorem 5 (Soundness of the Eiffel Contract Translator)**.**

$$\begin{array}{l}
\forall b : \mathit{BoolExp},\ t : \mathit{TypeFunc},\ e : \mathit{Expr},\ c : \mathit{CallRoutine},\ p : \mathit{Argument} : \\
(\mathit{wellF}_C\ b) \Rightarrow (\mathit{value}_C\ b\ h_1\ h_2\ s) = ((\nabla_C\ b)\ h_1\ h_2\ s)\ \ \mathit{and} \\
(\mathit{wellF}_T\ t) \Rightarrow (\mathit{value}_T\ t\ h_1\ h_2\ s) = ((\nabla_T\ t)\ h_1\ h_2\ s)\ \ \mathit{and} \\
(\mathit{wellF}_{\mathit{Exp}}\ e) \Rightarrow (\mathit{value}_{\mathit{Exp}}\ e\ h_1\ h_2\ s) = ((\nabla_{\mathit{Exp}}\ e)\ h_1\ h_2\ s)\ \ \mathit{and} \\
(\mathit{wellF}_{\mathit{Call}}\ c) \Rightarrow (\mathit{value}_{\mathit{Call}}\ c\ h_1\ h_2\ s) = ((\nabla_{\mathit{Call}}\ c)\ h_1\ h_2\ s)\ \ \mathit{and} \\
(\mathit{wellF}_{\mathit{Arg}}\ p) \Rightarrow (\mathit{value}_{\mathit{Arg}}\ p\ h_1\ h_2\ s) = ((\nabla_{\mathit{Arg}}\ p)\ h_1\ h_2\ s)
\end{array}$$

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 Poof 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, ∇_S and ∇_E for instructions and expressions respectively. The translation of expression is done with the same function ∇_E presented in Section 6.3. To be able to handle the Java exception handling, the translation function ∇_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:

$$\begin{array}{ll} \nabla_E & : \mathit{Precondition} \times \mathit{Expression} \times \mathit{Postcondition} \times \mathit{Label} \rightarrow \mathit{BytecodeProof} \\ \nabla_S & : \mathit{ProofTree} \times \mathit{Label} \times \mathit{Label} \times \mathit{Label} \times [\mathit{ExcType} \rightarrow \mathit{Label}] \rightarrow \mathit{BytecodeProof} \end{array}$$

ProofTree is a derivation in the logic presented in Section 3.3. The labels are: (1) l_{start} for the first label of the resulting bytecode; (2) l_{next} for the label after the resulting bytecode; (3) l_{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 $m : \mathit{ExcType} \rightarrow \mathit{Label}$, $m[T]$ 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:

$$T_{S_1} \equiv \frac{\mathit{Tree}_1}{\{\ P\ \}\quad s_1\quad \{\ Q_n, R_b, R_e\ \}} \qquad T_{S_2} \equiv \frac{\mathit{Tree}_2}{\{\ Q_n\ \}\quad s_2\quad \{\ R_n, R_b, R_e\ \}}$$

The definition of the translation is the following:

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

$$\nabla_S \left(T_{S_1},\ l_{start}, l_b, l_{break}, m\right)$$
$$\nabla_S \left(T_{S_2},\ l_b, l_{next}, l_{break}, m\right)$$

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_{while} be the following proof trees:

$$T_{S_1} \equiv \frac{Tree_1}{\left\{\ e\ \wedge\ I\ \right\}\ \ s_1\ \ \left\{\ I, Q_b, R_e\ \right\}}$$

$$T_{while} \equiv \frac{T_{S_1}}{\left\{\ I\ \right\}\ \ \texttt{while}\ (e)\ s_1\ \ \left\{\ (I \wedge \neg e) \vee\ Q_b, false, R_e\ \right\}}$$

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_{next}). The definition of the translation is the following:

$$\nabla_S \left(T_{while},\ l_{start}, l_{next}, l_{break},\ m\right) =$$

$$\{I\} \quad l_a : \mathsf{br}\ l_c$$
$$\nabla_S \left(T_{S_1},\ l_b, l_c, l_{next},\ m\right)$$
$$\nabla_E\ (\ I,\ e,\ (shift(I)\ \wedge\ s(0) = e)\ , l_c\ )$$
$$\{shift(I)\ \wedge\ s(0) = e\} \quad l_d :\ \mathsf{brtrue}\ l_b$$

The instruction 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

$$\mathit{shift}(I) \ \wedge \ s(0) = e$$

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}. Finally, the instruction brtrue establishes the preconditions of both possible successor instructions, namely $e \ \wedge \ I$ for the successor l_b (the first instruction of s_1), and $I \ \wedge \ \neg e$ for l_{next}. Therefore, the produced bytecode proof is valid.

8.1.4 Break Rule

The translation of the `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_{break}. Thus, the `break` instruction is translated using a br instruction. The definition of the translation is the following:

$$\nabla_S \left(\frac{}{\{\ P\ \} \ \ \texttt{break} \ \ \{\ \mathit{false}, P, \mathit{false}\ \}}, l_{start}, l_{next}, l_{break}, m \right)$$

$$\{P\} \quad l_{start} : \mathsf{br}\ l_{break}$$

To argue that the bytecode proof is valid, we have to show that the postcondition P implies the precondition of l_{break}. This can be proven using the hypothesis of the soundness theorem presented in Section 8.1.8.

8.1.5 Throw Rule

The `throw` instruction is translated using the CIL athrow instruction. The translation first pushes the expression e on top of the stack, and then it adds the instruction athrow. We assume that the instruction athrow takes the expression e on the top of the stack, then assigns it to the variable $excV$, 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:

$$\nabla_S\left(\frac{}{\left\{\ P[e/excV]\ \right\}\ \texttt{throw}\ e\ \left\{\ false, false, P\ \right\}}, l_{start}, l_{next}, l_{break},\ m\right) =$$

$$\begin{array}{l} \nabla_E(P[e/excV]\ ,\ e,\ \ (shift(P[e/excV])\ \wedge\ s(0) = e)\ ,\ l_{start}) \\ \{\ shift(P[e/excV])\ \wedge\ s(0) = e\ \}\ \ l_b : \mathsf{athrow} \end{array}$$

The translation for e establishes $shift(P[e/excV])\ \wedge\ s(0) = e$ 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:

$$T_{S_1} \equiv \frac{Tree_1}{\left\{\ P\ \right\}\ \ s_1\ \ \left\{\ Q_n, Q_b, Q\ \right\}}$$

$$T_{S_2} \equiv \frac{Tree_2}{\left\{\ Q'_e[e/excV]\ \right\}\ \ s_2\ \ \left\{\ Q_n, Q_b, R_e\ \right\}}$$

where

$$Q \equiv (\ (Q''_e\ \wedge\ \tau(excV) \npreceq T) \vee (Q'_e\ \wedge\ \tau(excV) \preceq T)\ )$$

Let R be the following postcondition:

$$R \equiv (R_e\ \vee\ (Q''_e\ \wedge\ \tau(excV) \npreceq T)\ )$$

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:

$$\nabla_S\left(\frac{T_{S_1} \qquad\qquad T_{S_2}}{\left\{\ P\ \right\}\ \texttt{try}\ s_1\ \texttt{catch}\ (T\ \ e)\ s_2\ \left\{\ Q_n, Q_b, R\ \right\}},\ l_{start}, l_{next}, l_{break}, m\right) =$$

```
.try {
    $\nabla_S\ \left(T_{S_1},\ l_{start},\ l_b,\ l_{break},\ m\left[\ \mathsf{T}\ \ \rightarrow l_c\ \right]\right)$
    $\{Q_n\}\quad l_b$ : leave $l_{next}$
}
catch System.Exception {
    $\left\{\begin{array}{l} shift(Q'_e)\ \wedge\ excV \neq null \\ \wedge\ \tau(excV) \preceq T\ \wedge\ s(0) = excV \end{array}\right\}\quad l_c$ : stloc $e$

    $\nabla_S\ \ (T_{S_2}, l_d,\ l_{next}, l_{break},\ m)$
    $\{Q_n\}\quad l_e$ : leave $l_{next}$
}
```

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:

$$T_{S_1} \equiv \frac{Tree_1}{\left\{\ P\ \right\}\ \ s_1\ \ \left\{\ Q_n, Q_b, Q_e\ \right\}} \qquad T_{S_2} \equiv \frac{Tree_2}{\left\{\ Q\ \right\}\ \ s_2\ \ \left\{\ R, R'_b, R'_e\ \right\}}$$

where

$$Q \equiv \left(\begin{array}{l} (Q_n \wedge \mathcal{X}Tmp = normal) \vee (Q_b \wedge \mathcal{X}Tmp = break) \vee \\ \left(Q_e[eTmp/excV] \wedge \mathcal{X}Tmp = exc \wedge eTmp = excV \right) \end{array} \right)$$

and

$$R \equiv \left(\begin{array}{l} (R'_n \wedge \mathcal{X}Tmp = normal) \vee (R'_b \wedge \mathcal{X}Tmp = break) \vee \\ (R'_e[eTmp/excV] \wedge \mathcal{X}Tmp = exc) \end{array} \right)$$

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 ∇_S, and then the instruction leave is added. In the finally block, the instruction s_2 is translated using ∇_S. The definition of the translation is:

$$\nabla_S \left(\frac{T_{S_1} \qquad T_{S_2}}{\left\{ P \right\} \ \texttt{try}\ s_1\ \texttt{finally}\ s_2 \ \left\{ R'_n, R'_b, R'_e \right\}}, l_{start}, l_{next}, l_{break}, m \right) =$$

.try $\{$

  $\nabla_S\ (T_{S_1},\ l_{start},\ l_b,\ l_{break},\ m)$

  $\{Q_n\}\quad l_b$: leave l_{next}

$\}$

.finally $\{$

  $\nabla_S\ (T_{S_2}, l_c,\ l_d, l_{break},\ m)$

  $\{R\}\quad l_d$: 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

$I_{l_{start}} ... I_{l_{end}}$, and their respective preconditions $E_{l_{start}} ... E_{l_{end}}$, and (3) the normal postcondition in the source logic implies the next precondition of the last generated instruction (if the last generated instruction is the last instruction of the method, we use the normal postcondition in the source logic), and (4) the break postcondition implies the precondition at the target label l_{break}, and (5) for all subtypes T of *Throwable* the exceptional postcondition in the source logic implies the precondition at the target label stored in the mapping function m ($m[T]$) but considering the value stored in the stack of the bytecode, then every bytecode specification holds ($\vdash \{E_l\}\ I_l$). The theorem is the following:

Theorem 6 (Soundness of the Java Instruction Translator)**.**

$$
\begin{array}{l}
\vdash \dfrac{\mathit{Tree}_1}{\left\{\ P\ \right\}\quad s_1\quad \left\{\ Q_n, Q_b, Q_e\ \right\}} \quad \wedge \\
(I_{l_{start}} ... I_{l_{end}}) = \nabla_S \left(\dfrac{\mathit{Tree}_1}{\left\{\ P\ \right\}\quad s_1\quad \left\{\ Q_n, Q_b, Q_e\ \right\}},\ l_{start}, l_{end+1}, l_{break}, m \right) \wedge \\
\left(Q_n \quad \Rightarrow \quad E_{l_{end+1}}\right) \quad \wedge \\
\left(Q_b \quad \Rightarrow \quad E_{l_{break}}\right) \quad \wedge \\
(\ \forall T : \mathit{Type} : T \preceq \mathit{Throwable} : (Q_e \ \wedge\ excV \neq null \ \wedge\ s(0) = excV) \quad \Rightarrow \quad E_{m[T]}) \\
\Rightarrow \\
\forall\ l\ \in\ l_{start}\ ...\ l_{end} : \vdash \{E_l\}\ I_l
\end{array}
$$

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:

```
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 ∇_S. The function ∇_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:

$$\begin{aligned}\nabla_S \quad &: \mathit{ProofTree} \times \mathit{Label} \times \mathit{Label} \times \mathit{Label} \times \mathit{List}[\mathit{Finally}] \times \\ &\quad \mathit{ExcTable} \rightarrow [\mathit{BytecodeProof} \times \mathit{ExcTable}]\end{aligned}$$

ProofTree is a derivation in the source logic. The three labels are: (1) l_{start} for the first label of the resulting bytecode; (2) l_{next} for the label after the resulting bytecode; (3) l_{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 ∇_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:

$$\begin{array}{ll} \mathit{ExcTable} & := \mathit{List}[\mathit{ExcTableEntry}] \\ \mathit{ExcTableEntry} & := [\mathit{Label}, \mathit{Label}, \mathit{Label}, \mathit{Type}] \end{array}$$

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 $[l_{start}, l_{end}, l_{targ}, T_2]$ if and only if $l_{start} \leq l < l_{end}$ and $T_1 \preceq T_2$. Control is then transferred to l_{targ}.

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:

$$T_{S_1} \equiv \frac{\mathit{Tree}_1}{\{\ P\ \}\quad s_1 \quad \{\ Q_n, R_b, R_e\ \}} \qquad T_{S_2} \equiv \frac{\mathit{Tree}_2}{\{\ Q_n\ \}\quad s_2 \quad \{\ R_n, R_b, R_e\ \}}$$

In the translation of T_{S_1}, the label l_{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 $T_{S1;S2}$ 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:

$$[B_{S_1}, et_1] = \nabla_S\ (T_{S_1}, l_{start}, l_b, l_{break}, f, et)$$
$$[B_{S_2}, et_2] = \nabla_S\ (T_{S_2}, l_b, l_{next}, l_{break}, f, et_1)$$

The translation is defined as follows:

$$\nabla_S \left(\frac{T_{S_1} \qquad T_{S_2}}{\{\ P\ \}\quad s_1; s_2 \quad \{\ R_n, R_b, R_e\ \}},\ l_{start}, l_{next}, l_{break}, f,\ et \right) = \\ [B_{S_1} + B_{S_2}\ ,\ et_2]$$

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.

8.2.3 While Rule

Let T_{S_1} be the following proof tree:

$$T_{S_1} \equiv \frac{\mathit{Tree}_1}{\{\ e\ \wedge\ I\ \}\quad s_1\quad \{\ I, Q_b, R_e\ \}}$$

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:

$$\begin{aligned}
b_{\mathsf{br}} &= \ \{I\} \quad l_a : \mathsf{br}\ l_c \\
[B_{S_1}, et_1] &= \ \nabla_S\,(T_{S_1},\ l_b, l_c, l_{next},\ \emptyset, et) \\
B_e &= \ \nabla_E\ (\ I,\ e,\ (\mathit{shift}(I)\ \wedge\ s(0) = e)\ , l_c\ ) \\
b_{\mathsf{brtrue}} &= \ \{\mathit{shift}(I)\ \wedge\ s(0) = e\} \quad l_d\ :\ \mathsf{brtrue}\ l_b
\end{aligned}$$

The definition of the translation is the following:

$$\nabla_S\left(\frac{T_{S_1}}{\{\ I\ \}\quad \texttt{while}\ (e)\ s_1\quad \{\ (I \wedge \neg e) \vee\ Q_b, \mathit{false}, R_e\ \}},\ l_{start}, l_{next}, l_{break},\ f,\ et\right) =$$
$$[\ b_{\mathsf{br}} + B_{S_1} + B_e + b_{\mathsf{brtrue}}\ ,\ et_1\ ]$$

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 $\mathit{shift}(I)\ \wedge\ s(0) = e$ 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 $e\ \wedge\ I$ for the successor l_b (the first instruction of B_{S_1}), and $I\ \wedge\ \neg e$ 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:

$$T_{S_1} \equiv \frac{\mathit{Tree}_1}{\{\ P\ \}\quad s_1\quad \{\ Q_n, Q_b, Q_e\ \}} \qquad T_{S_2} \equiv \frac{\mathit{Tree}_2}{\{\ Q\ \}\quad s_2\quad \{\ R, R'_b, R'_e\ \}}$$

where

$$Q \equiv \left(\begin{array}{l} (Q_n \wedge \mathcal{X}Tmp = normal) \ \vee (Q_b \wedge \mathcal{X}Tmp = break) \ \vee \\ (\ Q_e[eTmp/excV] \wedge \mathcal{X}Tmp = exc \wedge eTmp = excV \ ) \end{array} \right)$$

$$R \equiv \left(\begin{array}{l} (R'_n \wedge \mathcal{X}Tmp = normal) \ \vee \ (R'_b \wedge \mathcal{X}Tmp = break) \ \vee \\ (R'_e[eTmp/excV] \wedge \mathcal{X}Tmp = exc) \end{array} \right)$$

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 *getExcLines*. The signature of this function is as follows:

$$getExcLines : Label \times Label \times ExcTable \rightarrow ExcTable$$

Given two labels and an exception table et, *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:

$$\begin{array}{l} et_1 = et + [l_{start}, l_b, l_d, any] \\ et' = getExcLines(l_{start}, l_b, et_1) \end{array}$$

Let b_{br}, b_{stloc}, b_{ldloc}, and b_{athrow} be instructions specifications, and let B_{S_1}, B_{S_2}, and B'_{S_2} be bytecode proofs:

$$\begin{array}{rcll} [B_{S_1}, et_2] &=& \nabla_S \ (T_{S_1},\ l_{start},\ l_b,\ l_{break},\ [T_{S_2}\ , et'] + f,\ et_1) & \\ [B_{S_2}, et_3] &=& \nabla_S \ (T_{S_2},\ l_b,\ l_c,\ l_{break},\ f,\ et_2) & \\ b_{\mathsf{br}} &=& \{R'_n\} & l_c : \mathsf{br}\ l_{next} \\ b_{\mathsf{stloc}} &=& \left\{ \begin{array}{l} shift(Q_e) \wedge \\ excV \neq null \\ \wedge\ s(0) = excV \end{array} \right\} & l_d : \mathsf{stloc}\ eTmp \\ [B_{S'_2}, et_4] &=& \nabla_S \ (T_{S_2}, l_e, l_f, l_{break},\ f,\ et_3) & \\ b_{\mathsf{ldloc}} &=& \left\{ \ R'_n \ \vee \ R'_b \ \vee \ R'_e \ \right\} & l_f : \mathsf{ldloc}\ eTmp \\ b_{\mathsf{athrow}} &=& \left\{ \begin{array}{l} (R'_n \ \vee \ R'_b \ \vee \ R'_e) \\ \wedge\ s(0) = eTmp \end{array} \right\} & l_g : \mathsf{athrow} \end{array}$$

The translation is defined as follows:

$$\nabla_S\left(\frac{T_{S_1}\qquad T_{S_2}}{\left\{\ P\ \right\}\ \texttt{try}\ s_1\ \texttt{finally}\ s_2\ \left\{\ R'_n, R'_b, R'_e\ \right\}},\ l_{start}, l_{next}, l_{break},\ f,\ et\right) =$$
$$[\ B_{S_1} + B_{S_2} + b_{\mathsf{goto}} + b_{\mathsf{pop}} + B_{S'_2} + b_{\mathsf{pushv}} + b_{\mathsf{athrow}}\ ,\ et_4\ ]$$

It is easy to see that the instruction specifications b_{br}, b_{stloc}, b_{ldloc}, and b_{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 `break`, we use the following recursive function:

$$divide : ExcTable \times ExcTableEntry \times Label \times Label \rightarrow ExcTable$$

The definition of *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, *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 *divide* reached the expected entry. The formal definition of *divide* is the following:

$$
\begin{array}{lll}
\multicolumn{3}{l}{divide : ExcTable \times ExcTableEntry \ \times Label \ \times \ Label \rightarrow ExcTable} \\
\ divide : ([\,], e', l_s, l_e) & = [\ e'\ ] & \\
\ divide : (e : et, e', l_s, l_e) & = [\ l_{start},\ l_s,\ l_{targ},\ T_1\ ] \ + [\ l_e,\ l_{end},\ l_{targ},\ T_1\ ] + & \\
 & \quad divide(et, e', l_s, l_e) & \textbf{if } e \subseteq e' \ \wedge \ e \neq e' \\
 & \mid e : et & \textbf{if } e = e' \\
 & \mid e : divide(et, e', l_s, l_e) & \textbf{otherwise}
\end{array}
$$

where

$e \equiv [l_{start}, l_{end}, l_{targ}, T_1]$ and $e' \equiv [l'_{start}, l'_{end}, l'_{targ}, T_2]$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\subseteq : ExcTableEntry \times ExcTableEntry \rightarrow Boolean} \\
\subseteq : ([l_{start}, l_{end}, l_{targ}, T_1], [l'_{start}, l'_{end}, l'_{targ}, T_2]) & = true & \textbf{if } \ (l'_{st} \leq l_{st}) \ \wedge \ (l'_{end} \geq l_{end}) \\
 & \mid \ false & \textbf{otherwise}
\end{array}
$$

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_{break}. Let $f_i = [T_{F_i}, et'_i]$ denote the i-th element of the list f, where

$$T_{F_i} = \frac{Tree_i}{\{\ U^i\ \} \ \ s_i \ \ \{\ V^i\ \}}$$

and U^i and V^i have the following form, which corresponds to the Hoare rule for `try-finally`:

$$U^i \ \equiv \left\{ \begin{array}{l} (U^i_n \ \wedge \ \mathcal{X}Tmp = normal) \ \vee \ (U^i_b \ \wedge \ \mathcal{X}Tmp = break) \ \vee \\ (\ U^i_e[eTmp/excV] \ \wedge \ \mathcal{X}Tmp = exc \ \wedge \ eTmp = excV \ ) \end{array} \right\}$$

$$V^i \equiv \left\{ \left(\begin{array}{l} (V'^i_n \ \wedge \ \mathcal{X}Tmp = normal) \vee \\ (V'^i_b \ \wedge \ \mathcal{X}Tmp = break) \vee \\ (V'^i_e \ \wedge \ \mathcal{X}Tmp = exc) \end{array} \right), \ V^i_b, \ V^i_e \right\}$$

Let B_{Fi} be a *BytecodeProof* for T_{Fi} such that:

$$[B_{F_i}, et_{i+1}] = \ \nabla_S \ (\ T_{F_i}, l_{start+i}, l_{start+i+1}, l_{br}, f_{i+1}...f_k, divide(et_i, et'_i[0], l_{start+i}, l_{start+i+1}))$$

$$b_{\mathsf{br}} = \ \ \{B^k_b\} \quad l_{start+k+1} : \mathsf{br} \ l_{br}$$

The definition of the translation is the following:

$$\nabla_S \left(\frac{}{\{\ P\ \} \ \ \texttt{break} \ \ \{\ false, P, false\ \}}, l_{start}, l_{next}, l_{br}, f, et_0 \right)$$

$$= [\ B_{F_1} + B_{F_2} + ... + B_{F_k} + b_{\mathsf{br}}, et_k]$$

To argue that the bytecode proof is valid, we have to show that the postcondition of B_{F_i} 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 theses lemmas are presented in Appendix D

Lemma 3.
If

$$\nabla_S \left(\frac{Tree}{\{\ P\ \}\quad s\quad \{\ Q_n, Q_b, Q_e\ \}},\ l_a,\ l_{b+1},\ l_{break},\ f,\ et \right)\ =\ [(I_{l_a} ... I_{l_b}), et']$$

and $l_{start} \leq l_a < l_b \leq l_{end}$ *then,*
for every l_s, l_e : *Label such that* $l_b < l_s < l_e \leq l_{end}$*, and*
for every T : *Type such that* $T \preceq$ *Throwable* $\vee\ T \equiv$ *any, the following holds:*

$$et[l_{start}, l_{end}, T] = et'[l_s, l_e, T]$$

Lemma 4. *Let* r : *ExcTableEntry and* et' : *ExcTable be such that* $r \in et'$*.*
If et : *ExcTable and* l_s, l_e : *Label are such that* $et = divide(et', r, l_s, l_e)$*, 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 $I_{l_{start}} ... I_{l_{end}}$, their respective preconditions $E_{l_{start}} ... E_{l_{end}}$, 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 $T \preceq$ *Throwable* $\vee\ T \equiv$ *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 ($\vdash \{E_l\}\ I_l$).

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)**.**

$$
\left(
\begin{array}{l}
\vdash \dfrac{Tree}{\left\{\ P\ \right\}\ \ s_1\ \ \left\{\ Q_n, Q_b, Q_e\ \right\}} \equiv\ T_{S_1} \quad \wedge \\
[(I_{l_{start}} ... I_{l_{end}}), et] = \nabla_S\, (T_{S_1},\ l_{start}, l_{end+1}, l_{break}, f, et') \wedge \\
(\forall\ T : Type : (T \preceq Throwable \vee T \equiv any) : \\
\ (Q_e \wedge excV \neq null \wedge\ s(0) = excV)\ \Rightarrow\ E_{et'[l_{start}, l_{end}, T]}) \wedge \\
\big(Q_n \quad \Rightarrow \quad E_{l_{end+1}}\big) \quad \wedge \\
(Q_b\ \ \Rightarrow (f = \emptyset\ \Rightarrow\ (Q_b\ \ \Rightarrow\ \ E_{l_{break}}))\ \wedge \\
\qquad \left(
\begin{array}{l}
f \neq \emptyset \Rightarrow \forall i\ \in\ 1..k : \\
\left(
\begin{array}{l}
(\vdash \{U^i\}\ s_i\ \{V^i\} \quad \wedge \quad (V_b^i \Rightarrow U_b^{i+1})\ \wedge \\
(\ \forall T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (V_e^i\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \quad \Rightarrow \quad E_{et_i[T]}\ )\ \wedge \\
(Q_b \Rightarrow U_b^1)\ \wedge\ (V_b^k \Rightarrow E_{l_{break}})\ \wedge \\
(\forall et_i = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset) \wedge (et_i \subseteq et'))
\end{array}
\right)
\end{array}
\right)
\end{array}
\right)
$$

$$
\begin{array}{l}
\Rightarrow \\
\forall\ l\ \in\ l_{start}\ ...\ l_{end} : \vdash \{E_l\}\ I_l
\end{array}
$$

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:

$\{\ \mathbf{true}\ \}$	$IL001$: ldc 1
$\{s(0)\ =\ 1\}$	$IL002$: stloc b
$\{b\ =\ 1\}$	$IL003$: br $IL014$
// try block	
$\{b\ =\ 1\}$	$IL004$: newobj $Exception$
$\{b\ =\ 1\}$	$IL005$: athrow
// finally block	
$\{b\ =\ 1 \wedge excV \neq null\ \wedge\ s(0)\ =\ excV\}$	$IL007$: stloc $eTmp$
$\{b\ =\ 1\ \wedge\ eTmp\ =\ excV\}$	$IL008$: ldc 1
$\{b\ =\ 1\ \wedge\ s(0)\ =\ 1\}$	$IL009$: ldloc b
$\{b\ =\ 1\ \wedge\ s(1)\ =\ 1\ \wedge\ s(0)\ =\ b\}$	$IL010$: $\mathbf{binop}_{+}$
$\{b\ =\ 1\ \wedge\ s(0)\ =\ b+1\}$	$IL011$: stloc b
$\{b\ =\ 2\}$	$IL012$: br $IL017$
$\{\ \mathbf{false}\ \}$	$IL013$: ldloc $eTmp$
$\{\ \mathbf{false}\ \}$	$IL014$: throw
$\{b\ =\ 1\}$	$IL015$: ldc true
$\{b\ =\ 1\ \wedge\ s(0) = \mathbf{true}\ \}$	$IL016$: brtrue $IL004$
$\{b\ =\ 2\}$	$IL017$: ldc 1
$\{b\ =\ 2\ \wedge\ s(0) = 1\}$	$IL018$: ldloc b
$\{b\ =\ 2\ \wedge\ s(1) = 1\ \wedge\ s(0) = b\}$	$IL019$: $\mathbf{binop}_{+}$
$\{b\ =\ 2\ \wedge\ s(0) = 1 + b\}$	$IL020$: stloc b

Exception Table

From	*to*	*target*	*type*
$IL004$	$IL007$	$IL007$	*any*

Fig. 8.1: Bytecode Proof generated by the Java PTC from the Example Figure 3.14.

$$\frac{\begin{array}{ccc}\left\{\ \tau(\mathit{Current}) \prec \mathit{MATH}\ \wedge\ \mathit{true}\ \right\} & \mathit{MATH{:}foo} & \left\{\ b=3\ ,\ \mathit{false}\ \right\} \\ \left\{\ \tau(\mathit{Current}) = \mathit{MATH}\ \wedge\ \mathit{true}\ \right\} & \mathit{impl(MATH,\ foo)} & \left\{\ b=3\ ,\ \mathit{false}\ \right\}\end{array}}{\left\{\ \tau(\mathit{Current}) \preceq \mathit{MATH}\ \wedge\ \mathit{true}\ \right\}\ \ \mathit{MATH{:}foo}\ \ \left\{\ b=3\ ,\ \mathit{false}\ \right\}}$$

The first hypothesis:

$$\left\{\ \tau(\mathit{Current}) \prec \mathit{MATH}\ \wedge\ \mathit{true}\ \right\}\qquad \mathit{MATH{:}foo}\qquad \left\{\ b=3\ ,\ \mathit{false}\ \right\}$$

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:

$$\cfrac{\cfrac{\textit{proof Figure } 8.1}{\{\ \textit{true}\ \}\ \ \textit{body(MATH@foo)}\ \ \{\ b = 3\ ,\ \textit{false}\ \}}}{\{\ \textit{true}\ \}\ \ \textit{MATH@foo}\ \ \{\ b = 3\ ,\ \textit{false}\ \}}$$

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 *IL*004 − *IL*014. The body of the loop is between lines *IL*004 and *IL*016. The `break` translation yields at line *IL*012 a `br` instruction whose target is the end of the loop, *i.e.*, line *IL*017. Due to the execution of a `break` instruction, the code from *IL*017 to *IL*018 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.

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:

$$\forall s : \mathit{Source}, t : \mathit{Target} \;:\; \mathit{Comp}(s) = t \;\; \mathit{implies} \;\; \mathit{Sem}(s, t)$$

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.

Blech and Poetzsch-Heffter [24] have implemented a certified compiler for a subset of C. The compiler produces a machine-checkable proof in Isabelle. Blech and Gregoire [23] extended that work using the Coq theorem prover.

Proof-transforming compilers generate the bytecode proof directly, which leads to smaller certificates compared to certified compilers.

Translation Validation

Pnueli et al [104] have develop 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 $V : \mathit{Source} \times \mathit{Target} \rightarrow \mathit{Bool}$; 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 (*VCGen on Source*) takes the annotated program and its contracts, and generates the verification condition for the source program (*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 *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 `throw` and `try-catch` instructions. However, they do not consider `try-finally` instructions, which make the translation significantly more complex. Our translation supports `try-finally` and `break` instructions.

Pavlova et al. [103, 10] extends the aforementioned work to a subset of Java (which includes `try-catch`, `try-finally`, and `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 `finally` blocks by disallow-

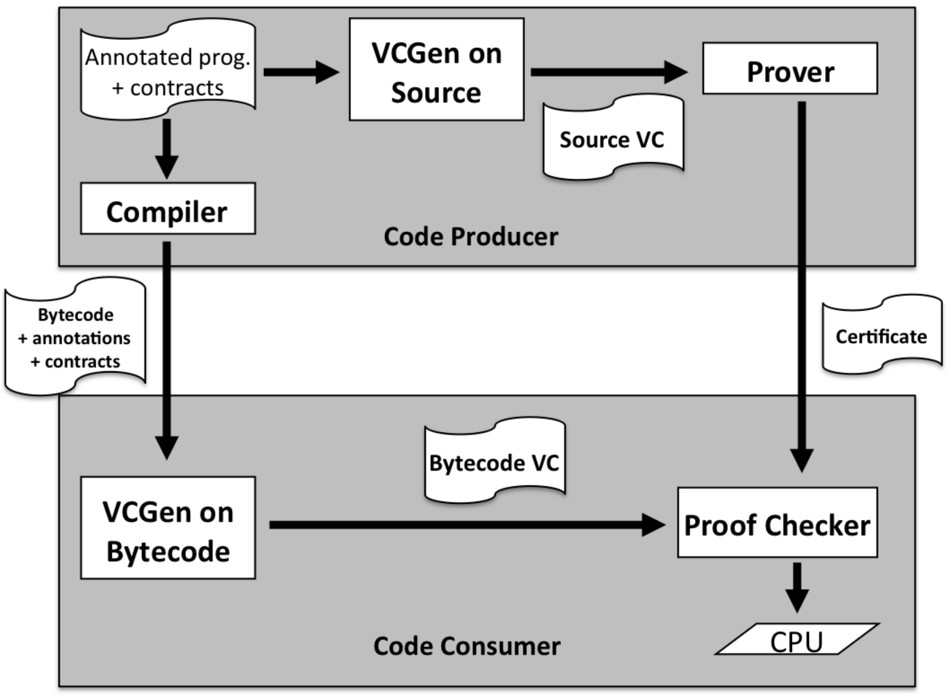

Fig. 8.2: General Architecture of Proof Preserving Compilation.

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[1]. 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.

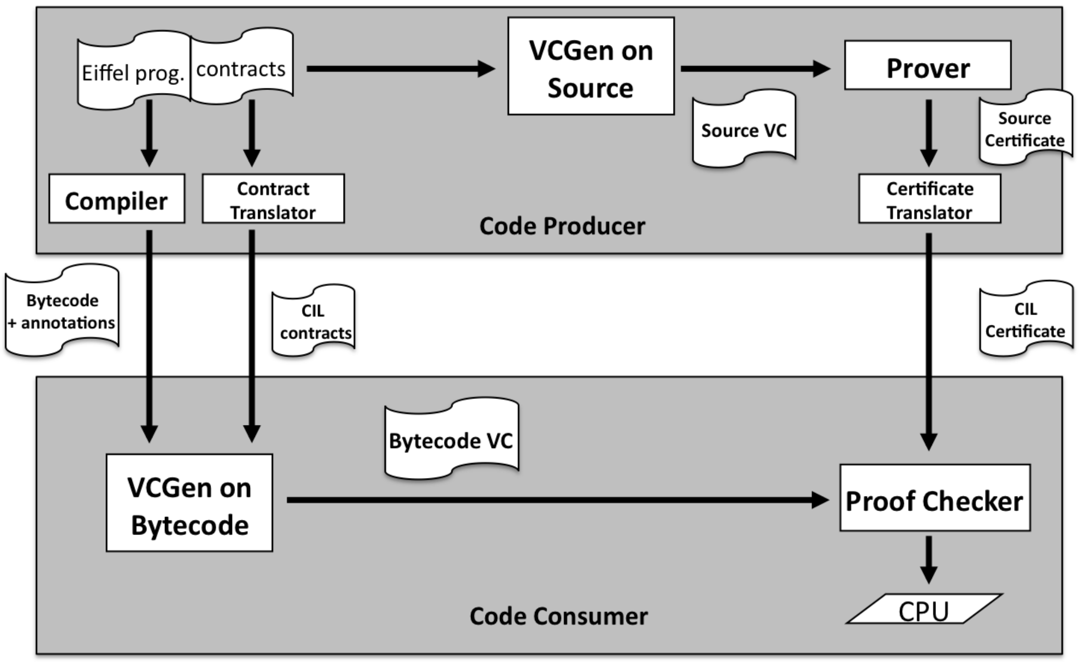

Fig. 8.3: General Architecture of Proof Preserving Compilation for Eiffel.

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.

Chapter 9

Implementation of the Proof-Transforming Compiler Scheme

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 [90]. 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 [45, 47, 58, 44]

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.

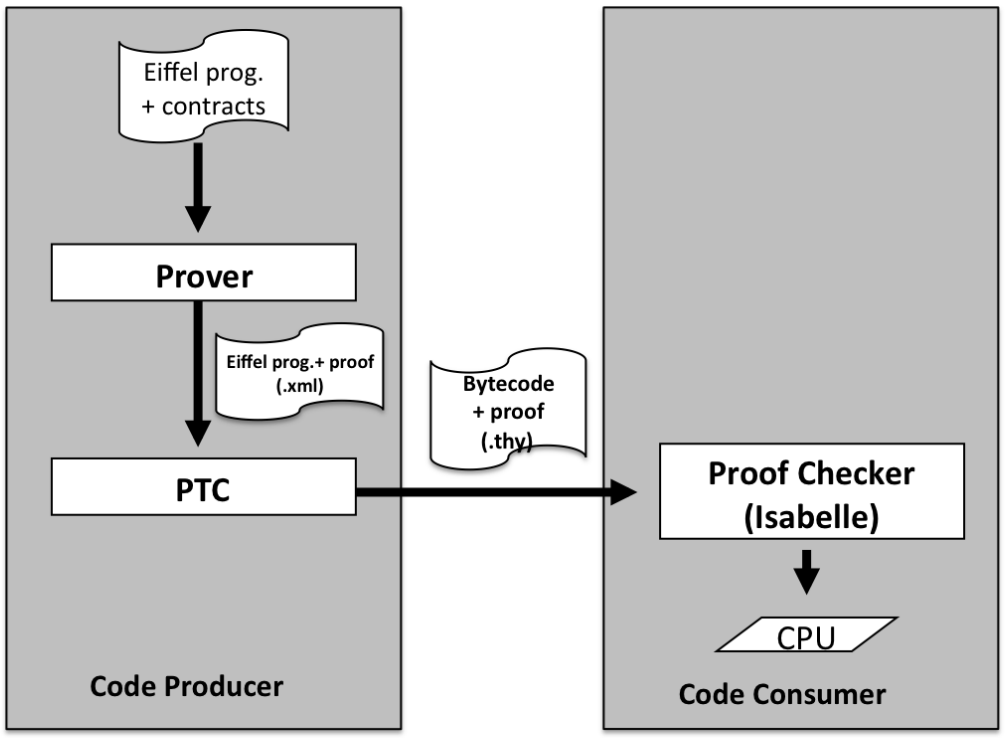

Fig. 9.1: General Architecture of the Implementation of the Proof-Transforming Compiler Schema.

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 generates 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

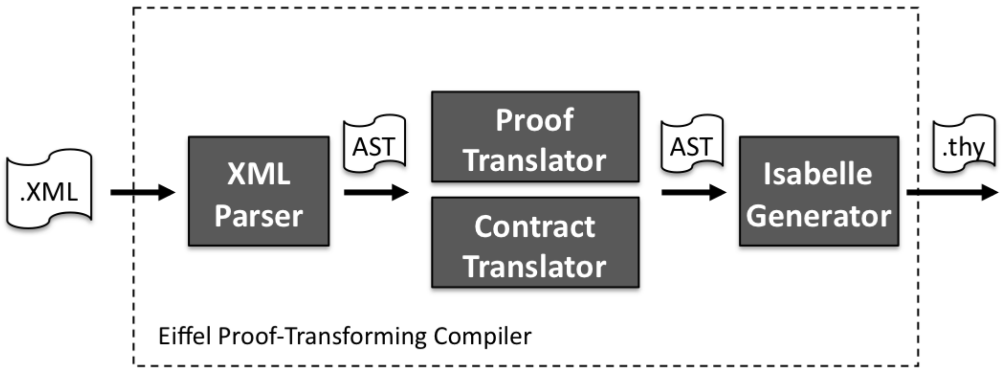

Fig. 9.2: General Architecture of the Implementation of the Eiffel Proof-Transforming Compiler.

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-independt rules (applied to method specifications).

The proof checker takes a list of instruction specifications. The rules for the CIL instruction specifications have the form:

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

To check the proof, the proof checker verifies that the implications $E_l \Rightarrow wp(I_l)$ 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* generates 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 add and ceq, branch instructions, load and store fields, object creation, method invocation, and throw instruction. Variables, field names, and method names are denoted by *VarId*, *FldId*, and *MethodId* respectively; while values, labels, and types are denoted by *Value*, *Label*, and *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 (*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:

$$\begin{aligned}
&\textbf{datatype } \mathit{ExcEntry} &&= \mathit{Label} \times \mathit{Label} \times \mathit{Label} \times \mathit{Type} \\
&\textbf{datatype } \mathit{ExcTable} &&= \textbf{list\_of } \mathit{ExcEntry}
\end{aligned}$$

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 $l_s \le l$ and $l < l_e$. The function *handlerEntry*, given an exception entry, returns the target label. The definitions of these functions are:

$$\begin{aligned}
&\mathit{is\_handled} : \mathit{ExcEntry} \times \mathit{Label} \times \mathit{Type} \rightarrow \mathit{bool} \\
&\quad \mathit{is\_handled}\ (l_s, l_e, l_t, T)\ l\ T' = \textbf{if } (l_s \le l \wedge l < l_e \wedge T' \preceq T) \textbf{ then } \mathit{true} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } \mathit{false}
\end{aligned}$$

$$\begin{aligned}
&\mathit{handlerEntry} : \mathit{ExcEntry} \rightarrow \mathit{Label} \\
&\quad \mathit{handlerEntry}\ (l_s,\ l_e,\ l_t,\ T) = l_t
\end{aligned}$$

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:

$$\begin{aligned}
&\mathit{is\_caught} : \mathit{ExcTable} \times \mathit{Label} \times \mathit{Type} \rightarrow \mathit{bool} \\
&\quad \mathit{is\_caught}\ [\,]\ l\ T = \mathit{false} \\
&\quad \mathit{is\_caught}\ (x\#xs)\ l\ T = \textbf{if } (\mathit{is\_handled}\ x\ l\ T) \textbf{ then } \mathit{true} \\
&\qquad\qquad\qquad\qquad\quad \textbf{else } (\mathit{is\_caught}\ xs\ l\ T)
\end{aligned}$$

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:

$$\begin{aligned}
&\mathit{handler} : \mathit{ExcTable} \times \mathit{Label} \times \mathit{Type} \rightarrow \mathit{Label} \\
&\quad \mathit{handler}\ [\,]\ l\ T = \mathit{arbitrary} \\
&\quad \mathit{handler}\ (x\#xs)\ l\ T = \textbf{if } (\mathit{is\_handled}\ x\ l\ T) \textbf{ then } (\mathit{handlerEntry}\ x) \\
&\qquad\qquad\qquad\qquad\quad \textbf{else } (\mathit{handler}\ xs\ l\ T)
\end{aligned}$$

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, 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 postcondition are boolean expression. Postconditions are a tuple of boolean expression: the first element stores the normal postcondition, and the second one the exceptional postcondition. The definition of CIL proofs is the following:

```
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 σ by $s(0)$. The definition of Wp is the following:

$$
\begin{array}{l}
\mathit{Wp} : \mathit{Instruction} \times \mathit{BoolExp} \times \mathit{CilProof} \times \mathit{BoolExp} \rightarrow \mathit{BoolExp}\\
\quad \mathit{Wp}\ \mathit{inst}\ \mathit{sucPre}\ \mathit{proof}\ \mathit{post} = \textbf{case}\ \mathit{inst}\ \textbf{of}\\
\qquad \begin{array}{lll}
\ \ \mathsf{ldloc}\ v & \rightarrow & (\mathit{unshift}(\mathit{subst}_x\ \mathit{sucPre}\ v))\\
\mid \mathsf{ldc}\ n & \rightarrow & (\mathit{unshift}(\mathit{subst}_c\ \mathit{sucPre}\ n))\\
\mid \mathsf{stloc}\ v & \rightarrow & (\lambda\ s : \mathit{Stack}\ \sigma : \mathit{State}\ (\mathit{subst}_\sigma(\mathit{shift}\ \mathit{sucPre})\ v)\\
\mid \mathit{binop} & \rightarrow & (\lambda s : \mathit{Stack}\ \sigma : \mathit{State}\ (\mathit{subst}_{c1}\ (\mathit{shift}\ \mathit{sucPre})\ (s(1)\ \mathit{binop}\ s(0))))\\
\mid \mathit{unop} & \rightarrow & (\lambda s : \mathit{Stack}\ \sigma : \mathit{State}\ (\mathit{subst}_{c1}\ (\mathit{shift}\ \mathit{sucPre})\ (\mathit{unnop}\ s(0))))\\
\mid \mathsf{br}\ l_2 & \rightarrow & (\mathit{PrecLabel}\ \mathit{proof}\ l_2)\\
\mid \mathsf{brtrue}\ l_2 & \rightarrow & (\lambda s : \mathit{Stack}\ \sigma : \mathit{State}\ (\neg s(0)\ \Rightarrow\ (\mathit{shift}\ \mathit{sucPre}))\ \wedge\\
 & & \qquad\qquad\qquad\qquad\quad s(0)\ \Rightarrow\ (\mathit{shift}(\mathit{PrecLabel}\ \mathit{proof}\ l_2)))\\
\mid \mathsf{brfalse}\ l_2 & \rightarrow & (\lambda s : \mathit{Stack}\ \sigma : \mathit{State}\ (s(0)\ \Rightarrow\ (\mathit{shift}\ \mathit{sucPre}))\ \wedge\\
 & & \qquad\qquad\qquad\qquad\quad \neg s(0)\ \Rightarrow\ (\mathit{shift}(\mathit{PrecLabel}\ \mathit{proof}\ l_2)))\\
\mid \mathsf{nop} & \rightarrow & \mathit{sucPre}\\
\mid \mathsf{ret} & \rightarrow & \mathit{post}
\end{array}
\end{array}
$$

where *PrecLabel* $(\mathit{proof}\ \ l)$ returns the precondition at label l.

9.2.5 Verification Condition Generator

The verification condition generator (*VCGen*) produces a proof obligation (a list of verification conditions). The verification condition (*VC*), and the proof obligation data types are defined as follows:

$$
\begin{array}{ll}
\textbf{datatype}\ \mathit{VC} & = \mathit{bool}\\
\textbf{datatype}\ \mathit{ProofObligation} & = \textbf{list\_of}\ \ \mathit{VC}
\end{array}
$$

The verification condition generator is defined by the functions *VCGenInst*, *VCGenMethod*, *VCGenClassBody*, *VCGen2*, and *VCGen*. The function *VCGenInst* generates verification conditions for instruction specifications. Except for callvirt and 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 callvirt and throw are generated by the functions *proof_oblig_call* and *proof_oblig_throw* respectively. The definition is:

$$\begin{array}{l}
\mathit{VCGenInst} : \mathit{InstSpec} \times \mathit{ExcTable} \times \mathit{CilBodyProof} \times \mathit{CilProof} \times \\
\qquad\qquad \mathit{BoolExp} \times \mathit{BoolExp} \rightarrow \mathit{VC} \\
\quad \mathit{VCGenInst}\ (\mathit{pre},\ l,\ \mathit{inst})\ \mathit{et}\ \mathit{cilProof}\ \mathit{prog}\ \mathit{postN}\ \mathit{postE}) \ = \mathbf{case}\ i\ \mathbf{of} \\
\qquad\qquad \mathbf{callvirt}\ T\ m\ e \rightarrow (\mathit{proof\_oblig\_call}\ \mathit{prog}\ T\ m\ e\ \mathit{pre}\ l\ \mathit{cilProof}\ \mathit{postN})) \\
\qquad\qquad |\ \mathbf{throw} \qquad\quad \rightarrow (\mathit{pre} \Rightarrow (\mathit{proof\_oblig\_throw}\ \mathit{et}\ l\ \mathit{postE}\ \mathit{cilProof})) \\
\qquad\qquad |\ \_ \qquad\qquad\quad \rightarrow (\mathit{pre} \Rightarrow (\mathit{Wp}\ \mathit{inst}\ l\ \mathit{cilProof}\ \mathit{postN}))
\end{array}$$

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:

$$\begin{array}{l}
\mathit{proof\_oblig\_call} : \mathit{CilProof} \times \mathit{Type} \times \mathit{MethodID} \times \mathit{Value} \times \mathit{BoolExp} \times \mathit{BoolExp} \rightarrow \mathit{bool} \\
\quad \mathit{proof\_oblig\_call}\ \mathit{prog}\ T\ m\ e\ \mathit{pre}\ \mathit{post} \ = \\
\qquad (\mathit{pre} \ \Rightarrow \ (\quad \lambda\ s : \mathit{Stack}\ \sigma : \mathit{State}. \\
\qquad\qquad\qquad \mathit{subst}_{\sigma 1}(\ \mathit{subst}_{\sigma}(\mathit{get\_prec}\ \mathit{prog}\ T\ m)\ \mathit{PARAM}\ )\ \mathit{THIS}))\ \wedge \\
\qquad\qquad\qquad (\ (\mathit{subst}_{\sigma}\ \ (\mathit{subst}_{pv}(\mathit{get\_post}\ \mathit{prog}\ T\ m)\ e)\ \ \mathit{RESULT})\ \ \Rightarrow\ \ \mathit{post})
\end{array}$$

where the function subst_{σ} substitutes a variable name in state σ by $s(0)$, the function $\mathit{subst}_{\sigma 1}$ substitutes a variable name in state σ by $s(1)$, subst_{pv} 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:

$$\begin{array}{l}
\mathit{handlerPost} : \mathit{ExcTable} \times \mathit{Label} \times \mathit{Type} \times \mathit{BoolExp} \times \mathit{CilProof} \rightarrow \mathit{BoolExp} \\
\quad \mathit{handlerPost}\ \mathit{et}\ l\ T\ \mathit{post}\ \mathit{proof} \ \ = \mathbf{if}\ (\mathit{is\_caught}\ \mathit{et}\ l\ T)\ \mathbf{then} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{PrecLabel}\ \mathit{proof}\ (\mathit{handler}\ \mathit{et}\ l\ T)) \\
\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \mathit{post} \\
\qquad\quad \textit{where PrecLabel (proof l) returns the precondition at label l}
\end{array}$$

$$\begin{array}{l}
\mathit{proof\_oblig\_throw} : \mathit{ExcTable} \times \mathit{Label} \times \mathit{BoolExp} \times \mathit{CilProof} \rightarrow \mathit{BoolExp} \\
\quad \mathit{proof\_oblig\_throw}\ \mathit{et}\ l\ \mathit{post}\ \mathit{proof} \ \ = (\lambda s.(\mathit{handlerPost}\ \mathit{et}\ l\ (\tau\ s(0)))\ \mathit{post}\ \mathit{proof})
\end{array}$$

The function *VCGenMethod* produces proof obligations for a method body proof. This function invokes *VCGenInst* per every instruction specification. The definition is:

$$
\begin{array}{l}
\mathit{VCGenMethod} : \mathit{CilBodyProof} \times \mathit{ExcTable} \times \mathit{CilBodyProof} \times \mathit{CilProof} \times \\
\qquad\qquad \mathit{BoolExp} \times \mathit{BoolExp} \rightarrow \mathit{ProofObligation} \\
\quad \mathit{VCGenMethod}\ \ [\,]\ \ \mathit{et\ cilProof\ prog\ postN\ postE} = [\mathit{true}] \\
\quad \mathit{VCGenMethod}\ (x\#xs)\ \mathit{et\ cilProof\ prog\ postN\ postE} = \\
\quad\ (\ \ \mathit{VCGenInst}\ \ x\ \mathit{et\ cilProof\ prog\ postN\ postE})\ \# \\
\quad\ (\mathit{VCGenMethod}\ xs\ \mathit{et\ cilProof\ prog\ postN\ postE})
\end{array}
$$

The function *VCGenClassBody* generates proof obligations for a class body, and the functions *VCGen*2 and *VCGen* produces proof obligations for CIL proofs. These functions are defined as follows:

$$
\begin{array}{l}
\mathit{VCGenClassBody} : \mathit{ClassBody} \ \times\ \mathit{CilProof} \ \rightarrow\ \mathit{ProofObligation} \\
\quad \mathit{VCGenClassBody}\ [\,]\ \mathit{prog}\ =\ [\mathit{true}] \\
\quad \mathit{VCGenClassBody}\ (x\#xs)\ \mathit{prog}\ = \\
\quad\ (\textbf{case}\ x\ \textbf{of}\ (m,\ \mathit{pre},\ \mathit{cilP},\ \mathit{et},\ \mathit{post}) \rightarrow \\
\quad\quad (\textbf{case}\ \mathit{post}\ \textbf{of}\ (\mathit{postN},\ \mathit{postE}) \rightarrow \\
\quad\quad\quad (\mathit{VCGenMethod\ cilP\ et\ cilP\ prog\ postN\ postE}) + (\mathit{VCGenClassBody\ xs\ prog})
\end{array}
$$

$$
\begin{array}{l}
\mathit{VCGen2} : \mathit{CilProof} \ \times\ \mathit{CilProof} \ \rightarrow\ \mathit{ProofObligation} \\
\quad \mathit{VCGen2}\ [\,]\ \mathit{prog} \qquad\qquad\qquad = [\mathit{true}] \\
\quad \mathit{VCGen2}\ ((\mathit{type}, \mathit{body})\#xs)\ \mathit{prog}\ = (\mathit{VCGenClassBody\ body\ prog}) + (\mathit{VCGen2}\ xs\ \mathit{prog})
\end{array}
$$

$$
\begin{array}{l}
\mathit{VCGen} : \mathit{CilProof} \rightarrow\ \mathit{ProofObligation} \\
\quad \mathit{VCGen}\ c\ \ = \mathit{VCGen2}\ c\ c
\end{array}
$$

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* : *CilProof, then*

$$\mathit{is\_program\_safe}(\mathit{VCGen}\ p)\ \textit{holds}.$$

The function *is_program_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:

$$\begin{array}{lcl} \mathit{is\_program\_safe} : \mathit{ProofObligation} & \rightarrow & \mathit{bool} \\ \quad \mathit{is\_program\_safe}\ [\,] & = & \mathit{true} \\ \quad \mathit{is\_program\_safe}\ (x\#xs) & = & \textbf{if}\ x\ \textbf{then}\ (\mathit{is\_program\_safe}\ xs)\ \textbf{else}\ \mathit{false}) \end{array}$$

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.

Example	**#Classes**	**#Routines**	**#lines in Eiffel**	**#lines source proof**	**#lines in Isabelle**
1. Boolean expressions	1	3	50	111	459
2. Attributes	2	3	83	167	1141
3. Conditionals	1	2	55	154	510
4. Loops	1	1	31	73	305
5. Bank Account	1	5	57	130	596
6. Sum Integers	1	1	35	126	358
7. Subtyping	3	5	41	117	756
8. Demo	18	28	548	1306	3718

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 *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 *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.

Isabelle Example	**#lines in Isabelle**	**Simplifier Proof Script**	**Optimized Proof Script**
1. Boolean expressions	459	9.4 sec.	6.9 sec.
2. Attributes	1141	7.6 sec.	6.2 sec.
3. Conditionals	510	9.3 sec.	7.8 sec.
4. Loops	305	18.1 sec.	7.2 sec.
5. Bank Account	596	16.8 sec.	8.6 sec.
6. Sum Integers	358	45.2 sec.	6.3 sec.
7. Subtyping	756	8.3 sec.	4.3 sec.
8. Demo	3718	5 min.	2 min.

Tab. 9.2: Experiments with the implemented PTC scheme: time expended to check the proof.

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

Type	Typical use
Instructions	s, s_1, s_2
Expressions	e, e_1, e_2
Virtual Routines	$T{:}m$, $S{:}m$
Routine implementations	T@m, S@m
Attributes (fields in Java)	$T@a$
Types	T, T', T_1, ..., T_n, S
States	$\sigma, \sigma', \sigma'', \sigma'''$
Object Store	\$, \$$'$
Precondition	P, P', P'', P'''
Normal Postcondition	Q_n, Q'_n, R_n
Postcondition for Exceptions	Q_e, Q'_e, Q''_e, R_e
Postcondition for Break	Q_b, R_b, R'_b
Loop Invariant	I
Sequent	$\mathcal{A}$, $\mathcal{A}_0$
Label (for examples bytecode)	*IL*001, ... *IL*026
Label (for the translation only)	$l_{start}, l_{exc}, l_{next}, l_{break}, l_{end}, l_{end+1}, l_a, l_b, ..., l_g$

Continued on next page

Continued from previous page

Type	Typical use
ProofTree (for source language only)	T_{S_1}, T_{S_2}, $Tree_i$, T_{sm}, T_{imp}, T_{tm},
	T_{body}, T_A, T_{A_0}
Bytecode Instruction at label l	I_l
Precondition at Label l (for Bytecode Logic)	E_l
ProofTree (for source language only)	T_{S_1}, T_{S_2}, $Tree_i$
ProofTree (for `finally` only)	T_{F_i}
List[*Finally*]	f, f_i
ExceptionTable	et_i
ExceptionTable (for `finally` only)	et_i'
BytecodeProof	B_{S_1}, B_{S_2}
InstrSpec	b_{pushc}, ..., b_{brtrue}
Label	l_{start}, ,
	l_b, l_c, ..., l_g

Appendix B

Soundness and Completeness Proof of the Logic

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 $\sigma, S \ \rightarrow_N \sigma', \mathit{normal}$ expresses that executing the instruction S in the state σ does not lead to more than N nested calls, and terminates normally in the state σ'. The transition $\sigma, S \ \rightarrow_N \sigma', \mathit{exc}$ expresses that executing the instruction S in the state σ does not lead to more than N nested calls, and terminates with an exception in the state σ'.

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:

$$\frac{\langle \sigma, s_1 \rangle \rightarrow_N \sigma', \mathit{normal} \qquad \langle \sigma', s_2 \rangle \rightarrow_N \sigma'', \chi}{\langle \sigma, s_1; s_2 \rangle \rightarrow_N \sigma'', \chi}$$

Routine Invocations. The routine invocation rule described in Section 3.1.3 is extended using the transition $\rightarrow_N$ as follows:

$$\frac{\begin{array}{c} T{:}m \textit{ is not a once routine} \\ \sigma(y) \neq \mathit{voidV} \qquad \langle \sigma[\mathit{Current} := \sigma(y), p := \sigma(e)],\ \mathit{body}(\mathit{impl}(\tau(\sigma(y)), m)) \rangle \rightarrow_N \sigma', \chi \end{array}}{\langle \sigma, x := y.T{:}m(e) \rangle \rightarrow_{N+1} \sigma'[x := \sigma'(\mathit{Result})], \chi} \tag{B.1}$$

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:

$$\frac{\begin{array}{c} T'@m = impl(\tau(\sigma(y)), m) \quad T'@m \text{ is a once routine} \\ \sigma(T'@m\_done) = false \\ \langle \sigma[T'@m\_done := true, Current := y, p := \sigma(e)],\ body(T'@m) \rangle \rightarrow_N \sigma', normal \end{array}}{\langle \sigma, x := y.S{:}m(e) \rangle \rightarrow_{N+1} \sigma'[x := \sigma'(Result)], normal}$$

$$\frac{\begin{array}{c} T@m = impl(\tau(\sigma(y)), m) \quad T@m \text{ is a once routine} \\ \sigma(T@m\_done) = false \\ \langle \sigma[T@m\_done := true, Current := y, p := \sigma(e)],\ body(T@m) \rangle \rightarrow_N \sigma', exc \end{array}}{\langle \sigma, x := y.S{:}m(e) \rangle \rightarrow_{N+1} \sigma'[T@m\_exc := true], exc}$$

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 $\models$). $\models \{\ P\ \}\quad s\quad \{\ Q_n\ ,\ Q_e\ \}$ *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
  - $\chi = normal \ \Rightarrow \sigma' \models Q_n$, *and*
  - $\chi = exc \ \Rightarrow \sigma' \models Q_e$
- *If s is the routine implementation $T@m$, then*
  for all $\sigma \models P : \langle \sigma, body(T@m) \rangle \rightarrow \sigma', \chi$ then
  - $\chi = normal \ \Rightarrow \sigma' \models Q_n$, *and*
  - $\chi = exc \ \Rightarrow \sigma' \models Q_e$
- *If s is the virtual routine $T{:}m$, then for all receivers x such that $\tau(x) = T$,*
  for all $\sigma \models P : \langle \sigma, body(imp(\tau(\sigma(x)), m)) \rangle \rightarrow \sigma', \chi$ then
  - $\chi = normal \ \Rightarrow \sigma' \models Q_n$, *and*

- $\chi = exc \;\Rightarrow \sigma' \models Q_e$

The definition of $\models \{\ P\ \}\quad s\quad \{\ Q_n\ ,\ Q_e\ \}$ (Definition 4) uses the transition $\rightarrow$. To handle recursive routine calls, we extend this definition using the transition $\rightarrow_N$ as follows:

Definition 5 (Triple $\models_N$). *$\models_N \{\ P\ \}\quad s\quad \{\ Q_n\ ,\ Q_e\ \}$ is defined as follows:*

- *If s is an instruction, then*
  $\models_N \{\ P\ \}\quad s\quad \{\ Q_n\ ,\ Q_e\ \}$ if only if:
  for all $\sigma \models P : \langle \sigma, s\rangle \rightarrow_N \sigma', \chi$ then
  - *$\chi = normal \;\Rightarrow \sigma' \models Q_n$, and*
  - *$\chi = exc \;\Rightarrow \sigma' \models Q_e$*
- *If s is the routine implementation $T@m$, then*
  $\models_0\ \{\ P\ \}\quad T@m\quad \{\ Q_n\ ,\ Q_e\ \}$ ***always holds****; and*
  $\models_{N+1} \{\ P\ \}\quad T@m\quad \{\ Q_n\ ,\ Q_e\ \}$ if only if $\models_N \{\ P\ \}\quad body(T@m)\quad \{\ Q_n\ ,\ Q_e\ \}$
- *If s is the virtual routine $T{:}m$, then for all receivers x such that $\tau(x) = T$*
  $\models_N \{\ P\ \}\quad T{:}m\quad \{\ Q_n\ ,\ Q_e\ \}$ if only if $\models_N \{\ P\ \}\quad imp(\tau(x), m)\quad \{\ Q_n\ ,\ Q_e\ \}$

The above definition presents the semantics for Hoare Triples with empty assumptions. The following definition introduces the semantics of sequent:

Definition 6 (Sequent Holds).

$\{P^1\}s_1\{Q^1_n\ ,\ Q^1_e\}, ..., \{P^j\}s_j\{Q^j_n\ ,\ Q^j_e\} \models \{P\}\quad s\quad \{Q_n\ ,\ Q_e\}$ 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

$$\models_N \{\ P\ \}\quad s\quad \{\ Q_n\ ,\ Q_e\ \}$$

Now, the theorems can be presented using the definition of sequent holds (Definition 6). The theorems are the followings:

Theorem 9 (Soundness Theorem).

$$\mathcal{A} \vdash \{\ P\ \}\quad s\quad \{\ Q_n\ ,\ Q_e\ \} \;\Rightarrow\; \mathcal{A} \models \{\ P\ \}\quad s\quad \{\ Q_n\ ,\ Q_e\ \}$$

Theorem 10 (Completeness Theorem).

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

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 $\models_N$).

$$\models \{\ P\ \}\quad s\quad \{\ Q_n\ ,\ Q_e\ \}\quad \textit{if only if}\ \ \forall N : \models_N\ \{\ P\ \}\quad s\quad \{\ Q_n\ ,\ Q_e\ \}$$

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 $\rightarrow_N$).

$$\langle \sigma,\ s_1\rangle \rightarrow \sigma', \chi\quad \textit{if only if}\quad \exists N : \langle \sigma,\ s_1\rangle \rightarrow_N \sigma', \chi$$

Lemma 8 (Monotone $\models_N$).

$$\models_{N+1} \{\ P\ \}\quad s\quad \{\ Q_n\ ,\ Q_e\ \}\quad \textit{implies}\quad \models_N \{\ P\ \}\quad s\quad \{\ Q_n\ ,\ Q_e\ \}$$

The proof of soundness runs by induction on the structure of the derivation tree for:

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

and the operational semantics.

Following, we present the soundness proof for the most interesting rules.

B.2.1 Assignment Axiom

We have to prove:

$$\vdash \left\{ \begin{array}{l} (\textit{safe}(e)\ \wedge\ P[e/x]) \vee \\ (\neg\textit{safe}(e)\ \wedge\ Q_e) \end{array} \right\}\ x\ :=\ e\ \ \{\ P\ ,\ Q_e\ \}\ \textit{implies}$$

$$\models \left\{ \begin{array}{l} (\textit{safe}(e)\ \wedge\ P[e/x]) \vee \\ (\neg\textit{safe}(e)\ \wedge\ Q_e) \end{array} \right\}\ x\ :=\ e\ \ \{\ P\ ,\ Q_e\ \}$$

Let P' be defined as $P' \triangleq (\textit{safe}(e)\ \wedge\ P[e/x])\ \vee\ (\neg\textit{safe}(e)\ \wedge\ Q_e)$.

Applying Definition 5, and Definition 6 to the consequence of the rule, we have to prove:

$\forall \sigma \models P' : \langle \sigma,\ x := e \rangle \rightarrow_N \sigma', \chi$ *then*

$$\begin{array}{rcl} \chi = normal & \Rightarrow & \sigma' \models P,\ and \\ \chi = exc & \Rightarrow & \sigma' \models Q_e \end{array}$$

We prove it doing case analysis on χ:
Case 1: χ = **exc**. By the definition of the operational semantics, we have:

$$\frac{\sigma(e) = exc}{\langle \sigma, x := e \rangle \rightarrow_N \sigma, exc}$$

Thus, we have to prove $\sigma \models Q_e$. Since $\sigma(e) = exc$, applying Lemma 1, we know $\sigma \models \neg safe(e)$. Since $\sigma \models P'$, and σ does not change, then $\sigma \models Q_e$.
Case 2: χ = **normal**. By the definition of the operational semantics, we get:

$$\frac{\sigma(e) \neq exc}{\langle \sigma, x := e \rangle \rightarrow_N \sigma[x := \sigma(e)], normal}$$

Thus, we have to prove $\sigma[x := \sigma(e)] \models P$. Since $\sigma(e) \neq exc$, applying Lemma 1, we know $\sigma \models safe(e)$. Since $\sigma \models P'$, then $\sigma \models safe(e)\ \wedge\ P[e/x]$. Applying Lemma 2, then $\sigma[x := \sigma(e)] \models P$.
□

B.2.2 Compound Rule

We have to prove:

$$\mathcal{A} \vdash \{\ P\ \}\quad s_1; s_2\quad \{\ R_n\ ,\ R_e\ \}\ \ implies\ \ \mathcal{A} \models \{\ P\ \}\quad s_1; s_2\quad \{\ R_n\ ,\ R_e\ \}$$

using the induction hypotheses:

$$\begin{array}{l} \mathcal{A} \vdash \{\ P\ \}\quad s_1\quad \{\ Q_n\ ,\ R_e\ \}\quad implies\ \ \mathcal{A} \models \{\ P\ \}\quad s_1\quad \{\ Q_n\ ,\ R_e\ \} \\ \mathcal{A} \vdash \{\ Q_n\ \}\quad s_2\quad \{\ R_n\ ,\ R_e\ \}\ \ implies\ \ \mathcal{A} \models \{\ Q_n\ \}\quad s_2\quad \{\ R_n\ ,\ R_e\ \} \end{array}$$

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:

$$for\ \ all\ \ N : \models_N \mathcal{A}_1, and..., \ \models_N \mathcal{A}_n\quad implies\quad \models_N \{\ P\ \}\quad s_1; s_2\quad \{\ R_n\ ,\ R_e\ \}$$

using the hypotheses:

$$\begin{array}{l} \textit{for all } N : \models_N \mathcal{A}_1, \textit{and...}, \models_N \mathcal{A}_n \ \ \textit{implies} \ \ \models_N \left\{ \ P \ \right\} \ \ s_1 \ \ \left\{ \ Q_n \ , \ R_e \ \right\} \\ \textit{for all } N : \models_N \mathcal{A}_1, \textit{and...}, \models_N \mathcal{A}_n \ \ \textit{implies} \ \ \models_N \left\{ \ Q_n \ \right\} \ \ s_2 \ \ \left\{ \ R_n \ , \ R_e \ \right\} \end{array}$$

Since the sequent $\mathcal{A}$ 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:

$$\begin{array}{l} \text{for all } \sigma \models P : \langle \sigma, \ s_1; s_2 \rangle \rightarrow_N \sigma'', \chi \ \ \textit{then} \\ \qquad\qquad\qquad \chi = \textit{normal} \ \Rightarrow \sigma'' \models R_n, \ \textit{and} \\ \qquad\qquad\qquad \chi = \textit{exc} \ \Rightarrow \sigma'' \models R_e \end{array} \tag{B.2}$$

using the hypotheses:

$$\begin{array}{l} \text{for all } \sigma \models P : \langle \sigma, \ s_1 \rangle \rightarrow_N \sigma', \chi \ \ \textit{then} \\ \qquad\qquad\qquad \chi = \textit{normal} \ \Rightarrow \sigma' \models Q_n, \ \textit{and} \\ \qquad\qquad\qquad \chi = \textit{exc} \ \Rightarrow \sigma' \models R_e \end{array} \tag{B.3}$$

and

$$\begin{array}{l} \text{for all } \sigma' \models Q_n : \langle \sigma', \ s_2 \rangle \rightarrow_N \sigma'', \chi \ \ \textit{then} \\ \qquad\qquad\qquad \chi = \textit{normal} \ \Rightarrow \sigma'' \models R_n, \ \textit{and} \\ \qquad\qquad\qquad \chi = \textit{exc} \ \Rightarrow \sigma'' \models R_e \end{array} \tag{B.4}$$

We prove it doing case analysis on χ:
Case 1: $\chi =$ **exc**. By the definition of the operational semantics for compound we get:

$$\frac{\langle \sigma, s_1 \rangle \rightarrow_N \sigma', \textit{exc}}{\langle \sigma, s_1; s_2 \rangle \rightarrow_N \sigma', \textit{exc}}$$

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:

$$\frac{\langle \sigma, s_1 \rangle \rightarrow_N \sigma', \textit{normal} \qquad \langle \sigma', s_2 \rangle \rightarrow_N \sigma'', \chi}{\langle \sigma, s_1; s_2 \rangle \rightarrow_N \sigma'', \chi}$$

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:

$$\chi = \textit{normal} \ \Rightarrow \sigma'' \models R_n, \ \textit{and} \ \chi = \textit{exc} \ \Rightarrow \sigma'' \models R_e$$

$\square$

B.2.3 Conditional Rule

We have to prove:

$$\begin{array}{l}\mathcal{A} \vdash \left\{ \; P \; \right\} \; \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end} \; \left\{ \; Q_n \; , \; Q_e \; \right\} \; \textit{implies} \\ \mathcal{A} \models \left\{ \; P \; \right\} \; \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end} \; \left\{ \; Q_n \; , \; Q_e \; \right\}\end{array}$$

using the induction hypotheses:

$$\begin{array}{l}\mathcal{A} \vdash \left\{ \; P \; \wedge \; e \; \right\} \; s_1 \; \left\{ \; Q_n \; , \; Q_e \; \right\} \quad \textit{implies} \;\; \mathcal{A} \models \left\{ \; P \; \wedge \; e \; \right\} \; s_1 \; \left\{ \; Q_n \; , \; Q_e \; \right\} \\ \mathcal{A} \vdash \left\{ \; P \; \wedge \; \neg e \; \right\} \; s_2 \; \left\{ \; Q_n \; , \; Q_e \; \right\} \; \textit{implies} \;\; \mathcal{A} \models \left\{ \; P \; \wedge \; \neg e \; \right\} \; s_2 \; \left\{ \; Q_n \; , \; Q_e \; \right\}\end{array}$$

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:

$$\begin{array}{l}\textit{for all } N : \models_N \mathcal{A}_1, \textit{and...}, \; \models_N \mathcal{A}_n \\ \textit{implies} \\ \models_N \left\{ \; P \; \right\} \; \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end} \; \left\{ \; Q_n \; , \; Q_e \; \right\}\end{array}$$

using the hypotheses:

$$\begin{array}{l}\textit{for all } N : \models_N \mathcal{A}_1, \textit{and...}, \; \models_N \mathcal{A}_n \; \textit{implies} \models_N \left\{ \; P \; \right\} \; s_1 \; \left\{ \; Q_n \; , \; Q_e \; \right\} \\ \textit{for all } N : \models_N \mathcal{A}_1, \textit{and...}, \; \models_N \mathcal{A}_n \; \textit{implies} \models_N \left\{ \; P \; \right\} \; s_2 \; \left\{ \; Q_n \; , \; Q_e \; \right\}\end{array}$$

Since the sequent $\mathcal{A}$ is the same in the hypotheses and the conclusions, and s_1 and s_2 are instructions, applying Definition 5 we have to prove:

$$\begin{array}{r}\forall \sigma \models P : \langle \sigma, \; \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end} \rangle \rightarrow_N \sigma', \chi \; \textit{then} \\ \chi = \textit{normal} \; \Rightarrow \sigma' \models Q_n, \; \textit{and} \\ \chi = \textit{exc} \; \Rightarrow \sigma' \models Q_e\end{array} \tag{B.5}$$

using the hypotheses:

$$\begin{array}{r}\forall \sigma \models (P \; \wedge \; e) : \langle \sigma, \; s_1 \; \rangle \rightarrow_N \sigma', \chi \; \textit{then} \\ \chi = \textit{normal} \; \Rightarrow \sigma' \models Q_n, \; \textit{and} \\ \chi = \textit{exc} \; \Rightarrow \sigma' \models Q_e\end{array} \tag{B.6}$$

and

$$\begin{array}{r}\forall \sigma \models (P \; \wedge \; \neg e) : \langle \sigma, \; s_2 \; \rangle \rightarrow_N \sigma', \chi \; \textit{then} \\ \chi = \textit{normal} \; \Rightarrow \sigma' \models Q_n, \; \textit{and} \\ \chi = \textit{exc} \; \Rightarrow \sigma' \models Q_e\end{array} \tag{B.7}$$

We prove this rule doing case analysis on $\sigma(e)$:

Case 1: $\sigma(\mathbf{e}) = \mathbf{true}$. If $\sigma(e) = \mathit{true}$ then by the definition of the operational semantics we get:

$$\frac{\langle \sigma, s_1 \rangle \rightarrow_N \sigma', \chi \qquad \sigma(e) = \mathit{true}}{\langle \sigma, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end} \rangle \rightarrow_N \sigma', \chi}$$

Then applying the first hypothesis (B.6) we prove $\chi = \mathit{normal} \ \Rightarrow \sigma' \models Q_n$, and $\chi = \mathit{exc} \ \Rightarrow \sigma' \models Q_e$.

Case 2: $\sigma(\mathbf{e}) = \mathbf{false}$. If $\sigma(e) = \mathit{false}$ then by the definition of the operational semantics we get:

$$\frac{\langle \sigma, s_2 \rangle \rightarrow_N \sigma', \chi \qquad \sigma(e) = \mathit{false}}{\langle \sigma, \texttt{if } e \texttt{ then } s_1 \texttt{ else } s_2 \texttt{ end} \rangle \rightarrow_N \sigma', \chi}$$

Then applying the second hypothesis (B.7) we prove $\chi = \mathit{normal} \ \Rightarrow \sigma' \models Q_n$, and $\chi = \mathit{exc} \ \Rightarrow \sigma' \models Q_e$.

$\square$

B.2.4 Check Axiom

We have to prove:

$$\vdash \ \left\{ \ P \ \right\} \ \texttt{check } e \texttt{ end} \ \left\{ \ (P \ \wedge \ e \ ) \ , \ (P \ \wedge \ \neg e \ ) \ \right\} \quad \mathit{implies}$$

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

Applying Definition 5 and Definition 6 to the consequence of the rule, we have to prove:

$$\begin{aligned} &\forall \sigma \models P : \langle \sigma, \ \texttt{check } e \texttt{ end} \rangle \rightarrow_N \sigma, \chi \ \mathit{then} \\ &\qquad\qquad \chi = \mathit{normal} \ \Rightarrow \sigma' \models (P \wedge e), \ \mathit{and} \\ &\qquad\qquad \chi = \mathit{exc} \ \Rightarrow \sigma' \models (P \wedge \neg e) \end{aligned} \tag{B.8}$$

To prove it, we do case analysis on $\sigma(e)$:

Case 1: $\sigma(\mathbf{e}) = \mathbf{true}$. By the definition of the operational semantics we have:

$$\frac{\sigma(e) = \mathit{true}}{\langle \sigma, \texttt{check } e \texttt{ end} \rangle \rightarrow_N \sigma, \mathit{normal}}$$

Since the state σ is unchanged then $\sigma \models P$. Furthermore, $\sigma(e) = \mathit{true}$ by this case analysis, then applying the definition of $\models$ we prove $\sigma \models (P \ \wedge \ e))$

Case 2: $\sigma(\mathbf{e}) = \mathbf{false}$. By the definition of the operational semantics we have:

$$\frac{\sigma(e) = \mathit{false}}{\langle \sigma, \texttt{check}\ e\ \texttt{end} \rangle \rightarrow_N \sigma, \mathit{exc}}$$

Similar to the above case, $\sigma \models P$ since the state is unchanged and $\sigma(e) = \mathit{false}$ by the case analysis. Then $\sigma \models (P\ \wedge\ \neg e)$ holds using the definition of $\models$.

□

B.2.5 Loop Rule

We have to prove:

$$\begin{array}{l} \mathcal{A} \vdash \{\ I\ \}\ \texttt{until}\ e\ \texttt{loop}\ s_1\ \texttt{end}\ \ \{\ (I\ \wedge\ e)\ ,\ R_e\ \}\ \mathit{implies} \\ \mathcal{A} \models \{\ I\ \}\ \texttt{until}\ e\ \texttt{loop}\ s_1\ \texttt{end}\ \ \{\ (I\ \wedge\ e)\ ,\ R_e\ \} \end{array}$$

using the induction hypotheses:

$$\begin{array}{l} \mathcal{A} \vdash \{\ \neg e\ \wedge\ I\ \}\ \ s_1\ \ \{\ I\ ,\ R_e\ \}\ \ \mathit{implies}\ \ \mathcal{A} \models \{\ \neg e\ \wedge\ I\ \}\ \ s_1\ \ \{\ I\ ,\ R_e\ \} \\ I\ \Rightarrow I' \end{array}$$

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:

$$\mathit{for\ all}\ N : \models_N \mathcal{A}_1, \mathit{and}..., \models_N \mathcal{A}_n\ \ \mathit{implies}\ \ \models_N \{\ P\ \}\ \texttt{from}\ ...\ \ \{\ (I\ \wedge\ e)\ ,\ R_e\ \}$$

using the hypothesis:

$$\mathit{for\ all}\ N : \models_N \mathcal{A}_1, \mathit{and}..., \models_N \mathcal{A}_n\ \ \mathit{implies}\ \ \models_N \{\ \neg e\ \wedge\ I\ \}\ \ s_1\ \ \{\ I\ ,\ R_e\ \}$$

Since the sequent $\mathcal{A}$ is the same in the hypothesis and the conclusions, and s_1 is an instruction, applying Definition 5 we have to prove:

$$\forall \sigma \models P : \forall \sigma \models P : \langle \sigma,\ \texttt{until}\ e\ \texttt{loop}\ s_1\ \texttt{end} \rangle \rightarrow_N \sigma', \chi\ \mathit{then}$$

$$\begin{array}{rcl} \chi = \mathit{normal} & \Rightarrow & \sigma' \models (I \wedge e),\ \mathit{and} \\ \chi = \mathit{exc} & \Rightarrow & \sigma' \models R_e \end{array}$$

using the hypothesis:

$$\begin{array}{l} \forall \sigma \models (\neg e\ \wedge I) : \langle \sigma,\ s_2\ \rangle \rightarrow_N \sigma', \chi\ \mathit{then} \\ \qquad\qquad \chi = \mathit{normal}\ \Rightarrow \sigma' \models I,\ \mathit{and} \\ \qquad\qquad \chi = \mathit{exc}\ \Rightarrow \sigma' \models R_e \end{array} \tag{B.9}$$

We prove this rule doing case analysis on $\sigma(e)$:

Case 1: $\sigma(\mathbf{e}) = \mathbf{true}$. By the operational semantics, we get:

$$\frac{\sigma(e) = \mathit{true}}{\langle \sigma, \texttt{until}\ e\ \texttt{loop}\ s_1\ \texttt{end}\rangle \rightarrow_N \sigma, \mathit{normal}}$$

Applying the hypothesis we know $\sigma \models I$ and $\sigma \models e$. Then by the definition of $\models$ we prove $\sigma \models (I\ \wedge\ e)$.

Case 2: $\sigma(\mathbf{e}) = \mathbf{false}$. Then we do case analysis on χ:

Case 2.a: $\chi = \mathbf{exc}$. By the definition of the operational semantics we have:

$$\frac{\sigma(e) = \mathit{false} \qquad \langle \sigma, s_1\rangle \rightarrow_N \sigma', \mathit{exc}}{\langle \sigma, \texttt{until}\ e\ \texttt{loop}\ s_1\ \texttt{end}\rangle \rightarrow_N \sigma', \mathit{exc}}$$

By the first hypothesis we know $\sigma \models I$. Then since $\sigma(e) = \mathit{false}$ and $\chi = \mathit{exc}$, we prove $\sigma' \models R_e$.

Case 2.b: $\sigma(\mathbf{e}) = \mathbf{false\ and}\ \chi = \mathbf{normal}$. By the definition of the operational semantics we have:

$$\frac{\begin{array}{c}\sigma(e) = \mathit{false} \qquad \langle \sigma, s_1\rangle \rightarrow_N \sigma', \mathit{normal}\\ \langle \sigma', \texttt{until}\ e\ \texttt{loop}\ s_1\ \texttt{end}\rangle \rightarrow_N \sigma'', \chi\end{array}}{\langle \sigma, \texttt{until}\ e\ \texttt{loop}\ s_1\ \texttt{end}\rangle \rightarrow_N \sigma'', \chi}$$

By the hypothesis we know $\sigma \models I$. Then since $\sigma(e) = \mathit{false}$, 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

$$\chi = \mathit{normal}\ \Rightarrow \sigma'' \models (I \wedge e),\ \mathit{and}\ \chi = \mathit{exc}\ \Rightarrow \sigma'' \models R_e$$

$\square$

B.2.6 Read Attribute Axiom

We have to prove:

$$\vdash \left\{ \begin{array}{l} (y \neq \mathit{Void}\ \wedge\ P[\$(\mathit{instvar}(y, T@a))/x]) \vee \\ (y = \mathit{Void}\ \wedge\ Q_e) \end{array} \right\}\ x := y.T@a\ \ \{\ P\ ,\ Q_e\ \}\quad \mathit{implies}$$

$$\models \left\{ \begin{array}{l} (y \neq \mathit{Void}\ \wedge\ P[\$(\mathit{instvar}(y, T@a))/x]) \vee \\ (y = \mathit{Void}\ \wedge\ Q_e) \end{array} \right\}\ x := y.T@a\ \ \{\ P\ ,\ Q_e\ \}$$

Applying Definition 5, and Definition 6 to the consequence of the rule, we have to prove:

$$\begin{array}{rl} \forall\sigma \models P' : \langle\sigma,\ x := y.T@a\rangle \to_N \sigma', \chi\ \mathit{then} & \\ \chi = \mathit{normal}\ \Rightarrow \sigma' \models P,\ \mathit{and} & \\ \chi = \mathit{exc}\ \Rightarrow \sigma' \models Q_e & \end{array} \tag{B.10}$$

where P' is defined as follows:

$$P' \equiv (y \neq \mathit{Void}\ \wedge\ P[\$(\mathit{instvar}(y, T@a))/x])\ \vee\ (y = \mathit{Void}\ \wedge\ Q_e)$$

To prove it, we do case analysis on χ:
Case 1: χ = **normal**. Applying the definition of the operational semantics we have:

$$\frac{\sigma(y) \neq \mathit{voidV}}{\langle\sigma, x := y.T@a\rangle \to_N \sigma[x := \sigma(\$)\ (\mathit{instvar}(\sigma(y), T@a))], \mathit{normal}}$$

Then applying Lemma 2 we get $\sigma \models P$.
Case 2: χ = **exc**. Applying the definition of the operational semantics:

$$\frac{\sigma(y) = \mathit{voidV}}{\langle\sigma, x := y.T@a\rangle \to_N \sigma, \mathit{exc}}$$

we get $\sigma \models Q_e$.
□

B.2.7 Write Attribute Axiom

We have to prove:

$$\vdash \left\{ \begin{array}{l} (y \neq \mathit{Void}\ \wedge\ P[\$\langle \mathit{instvar}(y, T@a) := e\rangle/\$])\ \vee \\ (y = \mathit{Void}\ \wedge\ Q_e) \end{array} \right\}\ y.T@a := e\ \ \{\ P\ ,\ Q_e\ \}\quad \mathit{implies}$$

$$\models \left\{ \begin{array}{l} (y \neq \mathit{Void}\ \wedge\ P[\$\langle \mathit{instvar}(y, T@a) := e\rangle/\$])\ \vee \\ (y = \mathit{Void}\ \wedge\ Q_e) \end{array} \right\}\ y.T@a := e\ \ \{\ P\ ,\ Q_e\ \}$$

Applying Definition 5, and Definition 6 to the consequence of the rule, we have to prove:

$$\begin{array}{rl} \forall\sigma \models P' : \langle\sigma,\ y.T@a := e\rangle \to_N \sigma', \chi\ \mathit{then} & \\ \chi = \mathit{normal}\ \Rightarrow \sigma' \models P,\ \mathit{and} & \\ \chi = \mathit{exc}\ \Rightarrow \sigma' \models Q_e & \end{array} \tag{B.11}$$

where P' is defined as follows:

$$P' \triangleq (y \neq Void \ \wedge \ P[\$\langle instvar(y, T@a) := e\rangle/\$]) \ \vee (y = Void \ \wedge \ Q_e)$$

To prove it, we do case analysis on χ:
Case 1: χ = **normal**. Applying the definition of the operational semantics we have:

$$\frac{\sigma(y) \neq voidV}{\langle \sigma, y.T@a := e\rangle \rightarrow_N \sigma[\$:= \sigma(\$)\langle instvar(\sigma(y), T@a) := \sigma(e)\rangle], normal}$$

Then applying Lemma 2 we get $\sigma \models P$.
Case 2: χ = **exc**. Applying the definition of the operational semantics:

$$\frac{\sigma(y) = voidV}{\langle \sigma, y.T@a := e\rangle \rightarrow_N \sigma, exc}$$

we get $\sigma \models Q_e$.
$\square$

B.2.8 Local Rule

We have to prove:

$$\begin{array}{l} \mathcal{A} \vdash \left\{ \ P \ \right\} \ \texttt{local} \ v_1 : T_1; \ ... \ v_n : T_n; \ s \ \ \left\{ \ Q_n \ , \ Q_e \ \right\} \ \ implies \\ \mathcal{A} \models \left\{ \ P \ \right\} \ \texttt{local} \ v_1 : T_1; \ ... \ v_n : T_n; \ s \ \ \left\{ \ Q_n \ , \ Q_e \ \right\} \end{array}$$

using the induction hypothesis:

$$\begin{array}{l} \mathcal{A} \vdash \left\{ \ P \ \wedge \ v_1 = init(T_1) \ \wedge \ ... \wedge \ v_n = init(T_n) \ \right\} \ \ s \ \ \left\{ \ Q_n \ , \ Q_e \ \right\} \ \ implies \\ \mathcal{A} \models \left\{ \ P \ \wedge \ v_1 = init(T_1) \ \wedge \ ... \wedge \ v_n = init(T_n) \ \right\} \ \ s \ \ \left\{ \ Q_n \ , \ Q_e \ \right\} \end{array}$$

Let $\mathcal{A}$ be the sequent $\mathcal{A} = \mathcal{A}_1, ..., \mathcal{A}_n$ where $\mathcal{A}_1, ..., \mathcal{A}_N$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

$$for \ all \ N : \models_N \mathcal{A}_1, and..., \models_N \mathcal{A}_n \ implies \models_N \left\{ \ P \ \right\} \ \texttt{local} \ v_1 : T_1; \ ... \ v_n : T_n; \ s \ \left\{ \ Q_n \ , \ Q_e \ \right\}$$

using the hypothesis:

$$\begin{array}{c} for \ all \ N : \models_N \mathcal{A}_1, and..., \models_N \mathcal{A}_n \ implies \\ \models_N \left\{ \ P \ \wedge \ v_1 = init(T_1) \ \wedge \ ... \wedge \ v_n = init(T_n) \ \right\} \ \ s \ \ \left\{ \ Q_n \ , \ Q_e \ \right\} \end{array}$$

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:

$$\forall \sigma \models P : \langle \sigma,\ \texttt{local}\ \ v_1 : T_1;\ ...\ v_n : T_n;\ s\rangle \rightarrow_N \sigma', \chi \ \textit{then}$$

$$\begin{aligned} \chi = \textit{normal} \quad &\Rightarrow \quad \sigma' \models Q_n,\ \textit{and} \\ \chi = \textit{exc} \quad &\Rightarrow \quad \sigma' \models Q_e \end{aligned}$$

using the hypothesis:

$$\begin{aligned} &\forall \sigma \models P' : \langle \sigma,\ s\ \rangle \rightarrow_N \sigma', \chi \ \textit{then} \\ &\qquad \chi = \textit{normal}\ \Rightarrow \sigma' \models Q_n,\ \textit{and} \\ &\qquad \chi = \textit{exc}\ \Rightarrow \sigma' \models Q_e \end{aligned} \tag{B.12}$$

where P' is defined as follows:

$$P' \triangleq P\ \wedge\ v_1 = \textit{init}(T_1)\ \wedge\ ... \wedge\ v_n = \textit{init}(T_n)$$

Applying the definition of the operational semantics for locals, we get:

$$\frac{\langle \sigma[v_1 := \textit{init}(T_1), ..., v_n := \textit{init}(T_n)],\ s\rangle \rightarrow_N \sigma', \textit{normal}}{\langle \sigma, \texttt{local}\ \ v_1 : T_1;\ ...\ v_n : T_n;\ s\rangle \rightarrow_N \sigma', \textit{normal}}$$

Then, applying Lemma 2 we know $\sigma \models P'$. Finally, applying the hypothesis (B.12) we prove:

$$\chi = \textit{normal}\ \Rightarrow \sigma' \models Q_n,\ \textit{and}\ \ \chi = \textit{exc}\ \Rightarrow \sigma' \models Q_e$$

□

B.2.9 Creation Rule

We have to prove:

$$\mathcal{A} \vdash \left\{\ P \left[\begin{array}{l} \textit{new}(\$, T)/\textit{Current}, \\ \$\langle T\rangle/\$, e/p \end{array} \right] \right\}\ \ x := \texttt{create}\ \{T\}.\textit{make}(e)\ \ \{\ Q_n[x/\textit{Current}]\ ,\ Q_e\ \}$$

implies

$$\mathcal{A} \models \left\{\ P \left[\begin{array}{l} \textit{new}(\$, T)/\textit{Current}, \\ \$\langle T\rangle/\$, e/p \end{array} \right] \right\}\ \ x := \texttt{create}\ \{T\}.\textit{make}(e)\ \ \{\ Q_n[x/\textit{Current}]\ ,\ Q_e\ \}$$

using the induction hypothesis:

$$\mathcal{A} \vdash \{\ P\ \}\ \ T : \textit{make}\ \ \{\ Q_n\ ,\ Q_e\ \}\ \ \textit{implies}\ \mathcal{A} \models \{\ P\ \}\ \ T : \textit{make}\ \ \{\ Q_n\ ,\ Q_e\ \}$$

Applying Definition 5 and Definition 6 to the consequence of the rule, we have to prove:

$\forall \sigma \models P' : \langle \sigma,\ x := \texttt{create}\ \{T\}.make(e)\rangle \rightarrow_{N+1} \sigma', \chi$ *then*

$$\begin{array}{rcl} \chi = normal & \Rightarrow & \sigma' \models Q_n[x/Current],\ and \\ \chi = exc & \Rightarrow & \sigma' \models Q_e \end{array}$$

where P' is defined as follows:

$$P' \triangleq P\left[\ new(\$, T)/Current,\ \$\langle T\rangle/\$, e/p\ \right]$$

using the hypothesis:
$\forall \sigma \models P : \langle \sigma,\ body(imp(T, make))\rangle \rightarrow_N \sigma', \chi$ *then*

$$\begin{array}{rcl} \chi = normal & \Rightarrow & \sigma' \models Q_n,\ and \\ \chi = exc & \Rightarrow & \sigma' \models Q_e \end{array}$$

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 $\sigma \models P[new(\$, T)/Current,\ \$\langle T\rangle/\$,\ e/p]$, then by Lemma 2, we know

$$\sigma[Current := new(\sigma(\$), T), \$:= \sigma(\$)\langle T\rangle, p := \sigma(e)] \models P$$

Applying the definition of the operational semantics we get

$$\begin{array}{ll} \chi = normal & \Rightarrow \sigma'[x := \sigma'(Current)] \models Q_n,\ and \\ \chi = exc & \Rightarrow \sigma' \models Q_e \end{array}$$

Using Lemma 2 we prove:

$$\chi = normal\ \Rightarrow \sigma' \models Q_n[x/Current],\ and\ \chi = exc\ \Rightarrow \sigma' \models Q_e$$

□

B.2.10 Rescue Rule

We have to prove:

$$\begin{array}{l} \mathcal{A} \vdash \left\{\ P\ \right\}\ s_1\ \texttt{rescue}\ s_2\ \left\{\ Q_n\ ,\ R_e\ \right\}\ implies \\ \mathcal{A} \models \left\{\ P\ \right\}\ s_1\ \texttt{rescue}\ s_2\ \left\{\ Q_n\ ,\ R_e\ \right\} \end{array}$$

using the induction hypotheses:

$$\begin{array}{l} \mathcal{A} \vdash \left\{\ I_r\ \right\}\ s_1\ \left\{\ Q_n\ ,\ Q_e\ \right\}\quad implies\quad \mathcal{A} \vdash \left\{\ I_r\ \right\}\ s_1\ \left\{\ Q_n\ ,\ Q_e\ \right\} \\ and \\ \mathcal{A} \vdash \left\{\ Q_e\ \right\}\ s_2\ \left\{\ (Retry \Rightarrow I_r)\ \wedge\ (\neg Retry \Rightarrow R_e)\ ,\ R_e\ \right\}\ implies \\ \mathcal{A} \models \left\{\ Q_e\ \right\}\ s_2\ \left\{\ (Retry \Rightarrow I_r)\ \wedge\ (\neg Retry \Rightarrow R_e)\ ,\ R_e\ \right\} \\ and \\ P\ \Rightarrow\ I_r \end{array}$$

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:

$$\textit{for all } N : \models_N \mathcal{A}_1, \textit{and...}, \ \models_N \mathcal{A}_n \textit{ implies} \models_N \left\{\ P\ \right\} \ \ s_1 \texttt{ rescue } s_2 \ \ \left\{\ Q_n\ ,\ R_e\ \right\}$$

using the hypotheses:

$$\begin{array}{l} \textit{for all } N : \models_N \mathcal{A}_1, \textit{and...}, \ \models_N \mathcal{A}_n \textit{ implies} \models_N \left\{\ I_r\ \right\} \ \ s_1 \ \ \left\{\ Q_n\ ,\ Q_e\ \right\} \\ \textit{for all } N : \models_N \mathcal{A}_1, \textit{and...}, \ \models_N \mathcal{A}_n \textit{ implies} \\ \qquad\qquad\qquad \models_N \left\{\ Q_e\ \right\} \ \ s_2 \ \ \left\{\ (\textit{Retry} \Rightarrow I_r) \ \wedge \ (\neg\textit{Retry} \Rightarrow R_e)\ ,\ R_e\ \right\} \end{array}$$

Since the sequent $\mathcal{A}$ is the same in the hypotheses and the conclusions, and s_1 and s_2 are instructions, applying Definition 5 we have to prove:
$\forall \sigma \models P : \langle \sigma, \ s_1 \texttt{ rescue } s_2 \rangle \rightarrow_N \sigma', \chi \textit{ then}$

$$\begin{array}{rcl} \chi = \textit{normal} & \Rightarrow & \sigma' \models Q_n, \ \textit{and} \\ \chi = \textit{exc} & \Rightarrow & \sigma' \models R_e \end{array}$$

using the hypotheses:

$$\begin{array}{l} \forall \sigma \models I_r : \langle \sigma, \ s_1 \ \rangle \rightarrow_N \sigma', \chi \textit{ then} \\ \qquad\qquad\qquad \chi = \textit{normal} \ \Rightarrow \sigma' \models Q_n, \ \textit{and} \\ \qquad\qquad\qquad\quad \chi = \textit{exc} \ \Rightarrow \sigma' \models Q_e \end{array} \tag{B.13}$$

and

$$\begin{array}{l} \forall \sigma \models Q_e : \langle \sigma, \ s_2 \ \rangle \rightarrow_N \sigma'', \chi \textit{ then} \\ \qquad\qquad\qquad \chi = \textit{normal} \ \Rightarrow \sigma'' \models ((\textit{Retry} \Rightarrow I_r) \ \wedge \ (\neg\textit{Retry} \Rightarrow R_e)), \ \textit{and} \\ \qquad\qquad\qquad\quad \chi = \textit{exc} \ \Rightarrow \sigma'' \models R_e \end{array} \tag{B.14}$$

We prove this rule doing case analysis on χ:

Case 1: $\chi =$ **normal**. Since s_1 terminates normally, by the definition of the operational semantics we get:

$$\frac{\langle \sigma, s_1 \rangle \rightarrow_N \sigma', \textit{normal}}{\langle \sigma, s_1 \texttt{ rescue } s_2 \rangle \rightarrow_N \sigma', \textit{normal}}$$

Then we can apply the first hypothesis (B.13) since $P \Rightarrow I_r$. Thus, we prove $\sigma' \models Q_n$.

Case 2: $\chi = \mathbf{exc}$. If s_1 triggers an exception, then by the operational semantics we have:

$$\langle \sigma,\ s_1 \rangle \rightarrow_N \sigma', exc$$

We have several cases depending if *Retry* evaluates to *true* or not, and if s_2 terminates normally or not:

Case 2.a: $\chi = \mathbf{exc}$. By the definition of the operational semantics we have:

$$\frac{\langle \sigma, s_1 \rangle \rightarrow_N \sigma', exc \qquad \langle \sigma', s_2 \rangle \rightarrow_N \sigma'', exc}{\langle \sigma, s_1 \texttt{ rescue } s_2 \rangle \rightarrow_N \sigma'', exc}$$

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: $\chi = \mathbf{normal}$. Here, we do case analysis on $\sigma''(Retry)$:

Case 2.b.1: $\sigma''(\mathbf{Retry}) = \mathbf{false}$. By the definition of the operational semantics we have:

$$\frac{\langle \sigma, s_1 \rangle \rightarrow_N \sigma', exc \qquad \langle \sigma', s_2 \rangle \rightarrow_N \sigma'', normal \qquad \neg\sigma''(Retry)}{\langle \sigma, s_1 \texttt{ rescue } s_2 \rangle \rightarrow_N \sigma'', exc}$$

By the first hypothesis (B.13), we prove $\sigma' \models Q_e$. Then, we can apply the second hypothesis (B.14) and we get $\sigma'' \models ((Retry \Rightarrow I_r) \wedge (\neg Retry \Rightarrow R_e))$. Since $\sigma''(Retry) = false$ then by the definition of $\models$ we prove $\sigma'' \models R_e$.

Case 2.b.2: $\sigma''(\mathbf{Retry}) = \mathbf{true}$. The definition of the operational semantics is:

$$\frac{\langle \sigma, s_1 \rangle \rightarrow_N \sigma', exc \qquad \langle \sigma', s_2 \rangle \rightarrow_N \sigma'', normal \qquad \sigma''(Retry) \qquad \langle \sigma'', s_1 \texttt{ rescue } s_2 \rangle \rightarrow_N \sigma''', \chi}{\langle \sigma, s_1 \texttt{ rescue } s_2 \rangle \rightarrow_N \sigma''', \chi}$$

Applying the first hypothesis (B.13), we prove $\sigma' \models Q_e$. Then, we can apply the second hypothesis (B.14), and we get $\sigma'' \models (Retry \Rightarrow I_r \wedge \neg Retry \Rightarrow R_e)$. Since $\sigma''(Retry) = true$ then by the definition of $\models$ we prove $\sigma'' \models I_r$. Now we can apply the induction hypothesis and we prove

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

□

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:

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

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

$$Q'_n \equiv \left\{ \begin{array}{l} T@m\_done\ \wedge\ \neg T@m\_exc \wedge \\ (Q_n \vee (\ P''\ \wedge\ Result = T\_M\_RES\ \wedge\ T@m\_result = T\_M\_RES\ )) \end{array} \right\}$$

$$Q'_e \equiv \{\ T@m\_done\ \wedge\ T@m\_exc\ \wedge\ (Q_e\ \vee P''')\ \}$$

To prove the once function rule, we have to prove:

$$\begin{array}{l} \mathcal{A} \vdash \{\ P\ \}\ \ T@m\ \ \{\ Q'_n\ ,\ Q'_e\ \}\quad implies \\ \mathcal{A} \models \{\ P\ \}\ \ T@m\ \ \{\ Q'_n\ ,\ Q'_e\ \} \end{array}$$

using the induction hypothesis:

$$\mathcal{A}, \{P\}\ \ T@m\ \{Q'_n,\ Q'_e\} \vdash$$

$$\left\{ \begin{array}{l} P'[false/T@m\_done] \wedge \\ T@m\_done \end{array} \right\}\ body(T@m)\ \ \{\ (\ Q_n\ \wedge\ T@m\_done\ )\ ,\ (\ Q_e\ \wedge\ T@m\_done\ )\ \}$$

implies

$$\mathcal{A}, \{P\}\ \ T@m\ \{Q'_n,\ Q'_e\} \models$$

$$\left\{ \begin{array}{l} P'[false/T@m\_done] \wedge \\ T@m\_done \end{array} \right\}\ body(T@m)\ \ \{\ (\ Q_n\ \wedge\ T@m\_done\ )\ ,\ (\ Q_e\ \wedge\ T@m\_done\ )\ \}$$

Let $\mathcal{A} = \mathcal{A}_1, ..., \mathcal{A}_n$ where $\mathcal{A}_1, ..., \mathcal{A}_N$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

$$for\ all\ N : \models_N \mathcal{A}_1, and...,\ \models_N \mathcal{A}_n\ implies \models_N \{\ P\ \}\ \ T@m\ \ \{\ Q'_n\ ,\ Q'_e\ \}$$

using the hypothesis:

$$\begin{array}{l} \textit{for all } N : \models_N \mathcal{A}_1, \textit{and}..., \models_N \mathcal{A}_n \textit{ and } \models_N \{P\} \;\; T@m \; \{Q'_n, \; Q'_e\} \\ \textit{implies} \\ \quad \models_N \left\{ \begin{array}{l} P'[\textit{false}/T@m\_done] \wedge \\ T@m\_done \end{array} \right\} \; \textit{body}(T@m) \; \left\{ (Q_n \wedge T@m\_done) \; , \; (Q_e \wedge T@m\_done) \right\} \end{array} \tag{B.15}$$

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

$$\models_N \mathcal{A}_1, \textit{and}..., \models_N \mathcal{A}_n \textit{ implies} \models_N \left\{ \; P \; \right\} \quad T@m \quad \left\{ \; Q'_n \; , \; Q'_e \; \right\}$$

we have to show

$$\models_{N+1} \mathcal{A}_1, \textit{and}..., \models_{N+1} \mathcal{A}_n \textit{ implies} \models_{N+1} \left\{ \; P \; \right\} \quad T@m \quad \left\{ \; Q'_n \; , \; Q'_e \; \right\}$$

Then, we can prove this as follows:

$$\begin{array}{ll} \models_{N+1} \mathcal{A}_1, \textit{and}..., \models_{N+1} \mathcal{A}_n & \\ & \textit{implies } [\textit{Lemma } 8] \\ \models_N \mathcal{A}_1, \textit{and}..., \models_N \mathcal{A}_n & \\ & \textit{applying induction hypothesis} \\ \models_N \left\{ \; P \; \right\} \quad T@m \quad \left\{ \; Q'_n \; , \; Q'_e \; \right\} & \end{array}$$

Using $\models_N \mathcal{A}_1, \textit{and}..., \models_N \mathcal{A}_n$, and $\models_N \left\{ \; P \; \right\} \quad T@m \quad \left\{ \; Q'_n \; , \; Q'_e \; \right\}$, we can apply the hypothesis (B.15), and we get:

$$\models_N \left\{ \begin{array}{l} P'[\textit{false}/T@m\_done] \wedge \\ T@m\_done \end{array} \right\} \; \textit{body}(T@m) \; \left\{ \; (Q_n \wedge T@m\_done) \; , \; (Q_e \wedge T@m\_done) \; \right\} \tag{B.16}$$

Since we know (B.16) holds, we can prove $\models_{N+1} \left\{ \; P \; \right\} \quad T@m \quad \left\{ \; Q'_n \; , \; Q'_e \; \right\}$ using the hypothesis (B.16). Applying Definition 5, we have to prove

$$\models_N \left\{ \; P \; \right\} \quad \textit{body}(T@m) \quad \left\{ \; Q'_n \; , \; Q'_e \; \right\}$$

assuming

$$\models_N \left\{ \begin{array}{l} P'[\mathit{false}/T@m\_done] \wedge \\ T@m\_done \end{array} \right\} \ \mathit{body}(T@m) \ \left\{ \ (Q_n \wedge T@m\_done) \ , \ (Q_e \wedge T@m\_done) \ \right\}$$

Then, applying Definition 5 ($\models_N$), we have to prove:
$\forall \sigma \models P : \langle \sigma, \ \mathit{body}(T@m) \rangle \rightarrow_N \sigma', \chi \ \mathit{then}$

$$\begin{array}{rcl} \chi = \mathit{normal} & \Rightarrow & \sigma' \models Q'_n, \ \mathit{and} \\ \chi = \mathit{exc} & \Rightarrow & \sigma' \models Q'_e \end{array}$$

using the hypothesis:
$\forall \sigma \models P'[\mathit{false}/T@m\_done] \wedge T@m\_done : \langle \sigma, \ \mathit{body}(T@m) \rangle \rightarrow_N \sigma', \chi \ \mathit{then}$

$$\begin{array}{rl} \chi = \mathit{normal} & \Rightarrow \sigma' \models Q_n \wedge \ T@m\_done, \ \mathit{and} \\ \chi = \mathit{exc} & \Rightarrow \sigma' \models Q_e \ \wedge \ T@m\_done \end{array} \tag{B.17}$$

We prove this with respect to the operational semantics of once routines (defined in page 38) by case analysis on χ and $T@m\_done$:

Case 1: σ(T@m_done) = false and χ = normal. By the definition of the operational semantics we have:

$$\frac{\begin{array}{c} T'@m = \mathit{impl}(\tau(\sigma(y)), m) \quad T'@m \ \mathit{is \ a \ once \ routine} \\ \sigma(T'@m\_done) = \mathit{false} \\ \langle \sigma[T'@m\_done := \mathit{true}, \mathit{Current} := y, p := \sigma(e)], \ \mathit{body}(T'@m) \rangle \rightarrow_N \sigma', \mathit{normal} \end{array}}{\langle \sigma, x := y.S{:}m(e) \rangle \rightarrow_{N+1} \sigma'[x := \sigma'(\mathit{Result})], \mathit{normal}}$$

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 $T@m = \mathit{imp}(T, m)$ and $\tau(\mathit{Current}) = T$. However, $T'@m = \mathit{imp}(\tau(y), m)$, and we now $\tau(y) = \tau(\mathit{Current})$, then we can conclude that $T@m = T'@m$.
Then, $\sigma \models P'[\mathit{false}/T@m\_done] \wedge \ T@m\_done$ because

$$\sigma[T'@m\_done := \mathit{true}, \mathit{Current} := y, p := \sigma(e)] \models P'$$

Now, we can apply the induction hypothesis B.17, and we get $\sigma' \models Q_n \ \wedge T@m\_done$. Since $Q_n \ \wedge T@m\_done \ \Rightarrow Q'_n$ then $\sigma' \models Q'_n$.

Case 2: σ(T@m_done) = false and χ = exc. By the definition of the operational semantics we have:

$$\frac{\begin{array}{c} T@m = impl(\tau(\sigma(y)), m) \quad T@m \text{ is a once routine} \\ \sigma(T@m\_done) = false \\ \langle \sigma[T@m\_done := true, Current := y, p := \sigma(e)],\ body(T@m)\rangle \rightarrow_N \sigma', exc \end{array}}{\langle \sigma, x := y.S{:}m(e)\rangle \rightarrow_{N+1} \sigma'[T@m\_exc := true], exc}$$

Applying a similar reasoning to Case 1, we know $T'@m = T@m$. Since:

$$\sigma \models P'[false/T@m\_done] \wedge \ T@m\_done$$

because $\sigma[T'@m\_done := true, Current := y, p := \sigma(e)] \models P'$, we can apply the induction hypothesis B.17, and we get $\sigma' \models Q_e \wedge T@m\_done$. Then $\sigma' \models Q'_e$ because $Q_e \wedge T@m\_done \Rightarrow Q'_e$

Case 3: σ(T@m_done) = true and χ = normal. The definition of the operational semantics is the following:

$$\frac{\begin{array}{c} T@m = impl(\tau(\sigma(y)), m) \quad T@m \text{ is a once routine} \\ \sigma(T@m\_done) = true \\ \sigma(T@m\_exc) = false \end{array}}{\langle \sigma, x := y.S{:}m(e)\rangle \rightarrow_N \sigma[x := \sigma(T@m\_result)], normal}$$

We know $T'@m = T@m$. Since $\sigma \models P$, and the state is unchanged except for the variable x, and $\sigma(T@m\_done) = true$ and $\sigma(T@m\_exc) = false$, then $\sigma \models P''$. Then $\sigma \models Q'_n$.

Case 4: σ(T@m_done) = true and χ = exc. By the definition of the operational semantics we have:

$$\frac{\begin{array}{c} T@m = impl(\tau(\sigma(y)), m) \quad T@m \text{ is a once routine} \\ \sigma(T@m\_done) = true \\ \sigma(T@m\_exc) = true \end{array}}{\langle \sigma, x := y.S{:}m(e)\rangle \rightarrow_N \sigma, exc}$$

We know $T'@m = T@m$. Since $\sigma \models P$, and the state is unchanged, and $\sigma(T@m\_done) = true$ and $\sigma(T@m\_exc) = true$, then $\sigma \models P'''$. Therefore, $\sigma \models Q'_e$.

This concludes the proof.

$\square$

B.2.12 Routine Implementation Rule

To prove this rule, we have to prove:

$$\mathcal{A} \vdash \{\ P\ \}\quad T@m\quad \{\ Q_n\ ,\ Q_e\ \}\quad \textit{implies}\quad \mathcal{A} \models \{\ P\ \}\quad T@m\quad \{\ Q_n\ ,\ Q_e\ \}$$

using the induction hypothesis:

$$\mathcal{A}, \{P\}\quad T@m\quad \{Q_n,\ Q_e\} \vdash\ \{\ P\ \}\quad \textit{body}(T@m)\quad \{\ Q_n\ ,\ Q_e\ \}\quad \textit{implies}$$

$$\mathcal{A}, \{P\}\quad T@m\quad \{Q_n,\ Q_e\} \models\ \{\ P\ \}\quad \textit{body}(T@m)\quad \{\ Q_n\ ,\ Q_e\ \}$$

Let $\mathcal{A}$ be the sequent $\mathcal{A} = \mathcal{A}_1, ..., \mathcal{A}_n$ where $\mathcal{A}_1, ..., \mathcal{A}_N$ are Hoare triples. By the semantics of sequents (Definition 6), we have to show:

$$\textit{for all } N : \models_N \mathcal{A}_1, \textit{and...},\ \models_N \mathcal{A}_n\ \textit{implies} \models_N \{\ P\ \}\quad T@m\quad \{\ Q_n\ ,\ Q_e\ \}$$

using the hypothesis:

$$\begin{aligned}&\textit{for all } N : \models_N \mathcal{A}_1, \textit{and...},\ \models_N \mathcal{A}_n,\ \textit{and}\ \models_N \{\ P\ \}\quad T@m\quad \{\ Q_n\ ,\ Q_e\ \}\\ &\quad \textit{implies}\\ &\qquad\qquad \models_N \{\ P\ \}\quad \textit{body}(T@m)\quad \{\ Q_n\ ,\ Q_e\ \}\end{aligned} \tag{B.18}$$

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

$$\models_N \mathcal{A}_1, \textit{and...},\ \models_N \mathcal{A}_n\ \textit{implies} \models_N \{\ P\ \}\quad T@m\quad \{\ Q_n\ ,\ Q_e\ \}$$

we have to show

$$\models_{N+1} \mathcal{A}_1, \textit{and...},\ \models_{N+1} \mathcal{A}_n\ \textit{implies} \models_{N+1} \{\ P\ \}\quad T@m\quad \{\ Q_n\ ,\ Q_e\ \}$$

Then, we can prove this as follows:

$$\begin{aligned}&\models_{N+1} \mathcal{A}_1, \textit{and...},\ \models_{N+1} \mathcal{A}_n\\ &\qquad\qquad\qquad \textit{implies}\ [\textit{Lemma } 8]\\ &\models_N \mathcal{A}_1, \textit{and...},\ \models_N \mathcal{A}_n\\ &\qquad\qquad\qquad \textit{applying induction hypothesis}\\ &\models_N \{\ P\ \}\quad T@m\quad \{\ Q_n\ ,\ Q_e\ \}\end{aligned}$$

Using $\models_N \mathcal{A}_1, \textit{and...},\ \models_N \mathcal{A}_n$, and $\models_N \{\ P\ \}\quad T@m\quad \{\ Q'_n\ ,\ Q'_e\ \}$, we can apply the hypothesis (B.18), and we get $\models_N\ \{\ P\ \}\quad \textit{body}(T@m)\quad \{\ Q_n\ ,\ Q_e\ \}$. Then, by Definition 5 we prove $\models_{N+1} \{\ P\ \}\quad T@m\quad \{\ Q_n\ ,\ Q_e\ \}$

□

B.2.13 Routine Invocation Rule

To prove this rule, we have to prove:

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

implies

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

using the induction hypothesis:

$$\mathcal{A} \vdash \{\ P\ \} \quad T{:}m \quad \{\ Q_n\ ,\ Q_e\ \} \quad \mathit{implies} \quad \mathcal{A} \models \{\ P\ \} \quad T{:}m \quad \{\ Q_n\ ,\ Q_e\ \}$$

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:

$$\begin{array}{l} \mathit{for\ all}\ N : \models_N \mathcal{A}_1, \mathit{and}..., \models_N \mathcal{A}_n \\ \mathit{implies} \\ \models_N \left\{ \begin{array}{l} (y \neq \mathit{Void} \wedge P[y/\mathit{Current}, e/p]) \vee \\ (y = \mathit{Void} \wedge Q_e) \end{array} \right\} \quad x := y.T{:}m(e) \quad \{\ Q_n[x/\mathit{Result}]\ ,\ Q_e\ \} \end{array}$$

using the hypothesis:

$$\mathit{for\ all}\ N : \models_N \mathcal{A}_1, \mathit{and}..., \models_N \mathcal{A}_n\ \mathit{implies} \models_N \{\ P\ \} \quad T{:}m \quad \{\ Q_n\ ,\ Q_e\ \}$$

Let P' be $(y \neq \mathit{Void} \wedge P[y/\mathit{Current}, e/p]) \vee\ (y = \mathit{Void} \wedge Q_e)$. Since the sequent $\mathcal{A}$ is the same in the hypothesis and the conclusion, applying Definition 5, we have to show:

$$\begin{array}{l} \text{for all } \sigma \models P' : \langle \sigma,\ x := y.T{:}m(e) \rangle \rightarrow_N \sigma'', \chi \ \ \mathit{then} \\ \qquad\qquad\qquad \chi = \mathit{normal}\ \Rightarrow \sigma'' \models Q_n[x/\mathit{Result}],\ \mathit{and} \\ \qquad\qquad\qquad\qquad \chi = \mathit{exc}\ \Rightarrow \sigma'' \models Q_e \end{array} \tag{B.19}$$

using the hypothesis:

$$\begin{array}{l} \text{for all } \sigma \models P : \langle \sigma,\ \mathit{body}(\mathit{imp}(\tau(\mathit{Current}), m)) \rangle \rightarrow_{N-1} \sigma', \chi \ \ \mathit{then} \\ \qquad\qquad\qquad \chi = \mathit{normal}\ \Rightarrow \sigma' \models Q_n,\ \mathit{and} \\ \qquad\qquad\qquad\qquad \chi = \mathit{exc}\ \Rightarrow \sigma' \models Q_e \end{array} \tag{B.20}$$

We do case analysis on $\sigma(y)$:
Case 1: $\sigma(\mathbf{y}) = \mathbf{void}$. By the operational semantics we have:

$$\frac{\begin{array}{c} T{:}m \textit{ is not a once routine} \\ \sigma(y) = voidV \end{array}}{\langle \sigma, x := y.T{:}m(e) \rangle \rightarrow_N \sigma, exc}$$

Then, $\sigma \models Q_e$ since $\sigma \models P$ and $\chi = exc$.
Case 2: $\sigma(\mathbf{y}) \neq \mathbf{void}$. The definition of the operational semantics is:

$$\frac{\begin{array}{cc} & T{:}m \textit{ is not a once routine} \\ \sigma(y) \neq voidV & \langle \sigma[Current := \sigma(y), p := \sigma(e)],\ body(impl(\tau(\sigma(y)), m)) \rangle \rightarrow_N \sigma', \chi \end{array}}{\langle \sigma, x := y.T{:}m(e) \rangle \rightarrow_{N+1} \sigma'[x := \sigma'(Result)], \chi}$$

Since $\sigma \models P'$, then applying Lemma 2, $\sigma \models P$. Then since $Current := \sigma(y)$, we can apply the induction hypothesis and Lemma 2 again, and we get

$$\chi = normal \ \Rightarrow \sigma'' \models Q_n[x/Result],\ and\ \chi = exc \ \Rightarrow \sigma'' \models Q_e$$

$\square$

B.2.14 Class Rule

We have to prove:

$$\begin{array}{llll} \mathcal{A} \vdash \{\ \tau(Current) \preceq T \ \wedge \ P\ \} & T{:}m & \{\ Q_n\ ,\ Q_e\ \} & implies \\ \mathcal{A} \models \{\ \tau(Current) \preceq T \ \wedge \ P\ \} & T{:}m & \{\ Q_n\ ,\ Q_e\ \} & \end{array}$$

using the induction hypotheses:

$$\begin{array}{llll} \mathcal{A} \vdash \{\ \tau(Current) = T \ \wedge \ P\ \} & imp(T,m) & \{\ Q_n\ ,\ Q_e\ \} & implies \\ \mathcal{A} \models \{\ \tau(Current) = T \ \wedge \ P\ \} & imp(T,m) & \{\ Q_n\ ,\ Q_e\ \} & \\ and & & & \\ \mathcal{A} \vdash \{\ \tau(Current) \prec T \ \wedge \ P\ \} & T{:}m & \{\ Q_n\ ,\ Q_e\ \} & implies \\ \mathcal{A} \models \{\ \tau(Current) \prec T \ \wedge \ P\ \} & T{:}m & \{\ Q_n\ ,\ Q_e\ \} & \end{array}$$

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:

$$\begin{array}{l} for\ all\ N : \models_N \mathcal{A}_1, and..., \ \models_N \mathcal{A}_n\ implies \\ \qquad \models_N \mathcal{A} \models \{\ \tau(Current) \preceq T \ \wedge \ P\ \} \quad T{:}m \quad \{\ Q_n\ ,\ Q_e\ \} \end{array}$$

using the hypotheses:

$$\begin{array}{l}
\textit{for all } N : \models_N \mathcal{A}_1, \textit{and}..., \models_N \mathcal{A}_n \textit{ implies} \\
\quad \models_N \mathcal{A} \models \{\ \tau(\textit{Current}) = T \ \wedge \ P \ \} \ \ \textit{imp}(T, m) \ \ \{\ Q_n \ , \ Q_e \ \} \\
\textit{and} \\
\textit{for all } N : \models_N \mathcal{A}_1, \textit{and}..., \models_N \mathcal{A}_n \textit{ implies} \\
\quad \models_N \mathcal{A} \models \{\ \tau(\textit{Current}) \prec T \ \wedge \ P \ \} \quad T{:}m \quad \{\ Q_n \ , \ Q_e \ \}
\end{array}$$

We prove it as follows:

$$\begin{array}{l}
\quad \mathcal{A} \vdash \{\ \tau(\textit{Current}) \preceq T \ \wedge \ P \ \} \quad T{:}m \quad \{\ Q_n \ , \ Q_e \ \} \\
\Rightarrow [\textit{definition of } \tau] \\
\quad \mathcal{A} \vdash \{\ (\tau(\textit{Current}) \prec T \ \vee \ \tau(\textit{Current}) = T) \ \wedge \ P \ \} \quad T{:}m \quad \{\ Q_n \ , \ Q_e \ \} \\
\Rightarrow [\textit{hypothesis}] \\
\quad \mathcal{A} \models \{\ \tau(\textit{Current}) = T \ \wedge \ P \ \} \ \ \textit{imp}(T, m) \ \ \{\ Q_n \ , \ Q_e \ \} \\
\quad \textit{and} \\
\quad \mathcal{A} \models \{\ \tau(\textit{Current}) \prec T \ \wedge \ P \ \} \quad T{:}m \quad \{\ Q_n \ , \ Q_e \ \} \\
\Rightarrow [\textit{definition of } \models_N] \\
\quad \mathcal{A} \models \{\ \tau(\textit{Current}) = T \ \wedge \ P \ \} \quad T{:}m \quad \{\ Q_n \ , \ Q_e \ \} \\
\quad \textit{and} \\
\quad \mathcal{A} \models \{\ \tau(\textit{Current}) \prec T \ \wedge \ P \ \} \quad T{:}m \quad \{\ Q_n \ , \ Q_e \ \} \\
\Rightarrow \\
\quad \mathcal{A} \models \{\ \tau(\textit{Current}) \preceq T \ \wedge \ P \ \} \quad T{:}m \quad \{\ Q_n \ , \ Q_e \ \}
\end{array}$$

□

B.2.15 Subtype Rule

We have to prove:

$$\begin{array}{l}
\mathcal{A} \vdash \{\ \tau(\textit{Current}) \preceq S \ \wedge \ P \ \} \ \ T{:}m \ \ \{\ Q_n \ , \ Q_e \ \} \textit{ implies} \\
\mathcal{A} \models \{\ \tau(\textit{Current}) \preceq S \ \wedge \ P \ \} \ \ T{:}m \ \ \{\ Q_n \ , \ Q_e \ \}
\end{array}$$

using the induction hypotheses:

$$\begin{array}{l}
\mathcal{A} \vdash \{\ P \ \} \ \ S{:}m \ \ \{\ Q_n \ , \ Q_e \ \} \ \ \textit{implies } \mathcal{A} \models \{\ P \ \} \ \ S{:}m \ \ \{\ Q_n \ , \ Q_e \ \} \\
S \preceq T
\end{array}$$

We have to prove:

$$\begin{array}{l}
\quad \mathcal{A} \models \{\ \tau(\textit{Current}) \preceq S \ \wedge \ P \ \} \ \ T{:}m \ \ \{\ Q_n \ , \ Q_e \ \} \\
\textit{iff} [\textit{definition of } \models_N] \\
\quad \mathcal{A} \models \{\ \tau(\textit{Current}) \preceq S \ \wedge \ P \ \} \ \ \textit{imp}(\tau(\textit{Current}), m) \ \ \{\ Q_n \ , \ Q_e \ \}
\end{array}$$

Since $\tau(\mathit{Current}) \preceq S$ and from the hypothesis we know $\mathcal{A} \models \{\ P\ \}\ \ S{:}m\ \ \{\ Q_n\ ,\ Q_e\ \}$, then applying the induction hypothesis we prove:

$$\mathcal{A} \models \{\ \tau(\mathit{Current}) \preceq S\ \wedge\ P\ \}\ \ T{:}m\ \ \{\ Q_n\ ,\ Q_e\ \}$$

□

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 : $\models_N \mathcal{A}$ *implies* $\models_N \mathcal{A}$, which is *true*.

□

False-axiom

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:

$$\mathit{for\ all}\ N : \models_N \mathcal{A}_1, \mathit{and}..., \models_N \mathcal{A}_n\ \mathit{implies}\ \models_N \{\ \mathit{false}\ \}\ \ s\ \ \{\ \mathit{false}\ ,\ \mathit{false}\ \}$$

This holds by the definition of $\models_N$.

□

Assumpt-intro-rule

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:

$$\mathit{for\ all}\ N : \models_N \mathcal{A}_1, \mathit{and}..., \models_N \mathcal{A}_n, \mathit{and}\ \mathbf{A_0}\ \mathit{implies}\ \models_N \mathbf{A}$$

using the hypothesis:

$$\mathit{for\ all}\ N : \models_N \mathcal{A}_1, \mathit{and}..., \models_N \mathcal{A}_n\ \mathit{implies}\ \models_N \mathbf{A}$$

This holds by the hypothesis.

□

Assumpt-elim-rule

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:

$$\textit{for all } N : \models_N \mathcal{A}_1, \textit{and}..., \ \models_N \mathcal{A}_n \textit{ implies } \ \models_N \mathbf{A}$$

using the hypotheses:

$$\begin{array}{l} \textit{for all } N : \models_N \mathcal{A}_1, \textit{and}..., \ \models_N \mathcal{A}_n \textit{ implies } \ \models_N \mathbf{A_0} \\ \textit{for all } N : \models_N \mathcal{A}_1, \textit{and}..., \ \models_N \mathcal{A}_n, \ \textit{and } \mathbf{A_0} \textit{ implies } \ \models_N \mathbf{A} \end{array}$$

We prove it as follows:

$$\begin{array}{l} \quad \models_N \mathcal{A}_1, \textit{and}..., \ \models_N \mathcal{A}_n \quad (1) \\ \Rightarrow [\textit{applying the first hypothesis}] \\ \quad \models_N \mathbf{A_0} \quad (2) \\ \Rightarrow [\textit{applying second hypothesis to } (1) \textit{ and } (2)] \\ \quad \models_N \mathbf{A} \end{array}$$

□

Strength Rule

We have to prove:

$$\mathcal{A} \vdash \{ \ P' \ \} \quad s_1 \quad \{ \ Q_n \ , \ Q_e \ \} \quad \textit{implies} \ \ \mathcal{A} \models \{ \ P' \ \} \quad s_1 \quad \{ \ Q_n \ , \ Q_e \ \}$$

using the induction hypotheses:

$$\begin{array}{l} \mathcal{A} \vdash \{ \ P \ \} \quad s_1 \quad \{ \ Q_n \ , \ Q_e \ \} \quad \textit{implies} \ \mathcal{A} \models \{ \ P \ \} \quad s_1 \quad \{ \ Q_n \ , \ Q_e \ \} \\ \textit{and} \\ P' \Rightarrow P \end{array}$$

Applying the definition of $\models_N$, we have to prove:

$$\begin{array}{rl} \text{for all } \sigma \models P' : \langle \sigma, \ s_1 \rangle \rightarrow_N \sigma'', \chi \ \ \textit{then} & \\ \chi = \textit{normal} & \Rightarrow \sigma'' \models Q_n, \ \textit{and} \\ \chi = \textit{exc} & \Rightarrow \sigma'' \models Q_e \end{array}$$

Since $P' \Rightarrow P$, then we get $\sigma \models P$, then by hypothesis we prove:

$$\mathcal{A} \models \{ \ P' \ \} \quad s_1 \quad \{ \ Q_n \ , \ Q_e \ \}$$

□

Weak Rule

We have to prove:

$$\mathcal{A} \vdash \{\ P\ \}\quad s_1 \quad \{\ Q'_n\ ,\ Q'_e\ \}\ \ \textit{implies}\ \ \mathcal{A} \models \{\ P\ \}\quad s_1 \quad \{\ Q'_n\ ,\ Q'_e\ \}$$

using the induction hypotheses:

$$\begin{array}{l} \mathcal{A} \vdash \{\ P\ \}\quad s_1 \quad \{\ Q_n\ ,\ Q_e\ \}\ \ \textit{implies}\ \mathcal{A} \models \{\ P\ \}\quad s_1 \quad \{\ Q_n\ ,\ Q_e\ \} \\ \textit{and} \\ Q_n \Rightarrow Q'_n \\ Q_e \Rightarrow Q'_e \end{array}$$

Applying the definition of $\models_N$, we have to prove:

$$\begin{array}{r} \text{for all } \sigma \models P : \langle \sigma,\ s_1 \rangle \rightarrow_N \sigma'', \chi\ \ \textit{then} \\ \chi = \textit{normal}\ \Rightarrow \sigma'' \models Q'_n,\ \textit{and} \\ \chi = \textit{exc}\ \Rightarrow \sigma'' \models Q'_e \end{array}$$

Applying the hypothesis we get:

$$\begin{array}{ll} \chi = \textit{normal} & \Rightarrow \sigma'' \models Q_n,\ \textit{and} \\ \chi = \textit{exc} & \Rightarrow \sigma'' \models Q_e \end{array}$$

Since $Q_n \Rightarrow Q'_n$ and $Q_e \Rightarrow Q'_e$ then we get

$$\begin{array}{ll} \chi = \textit{normal} & \Rightarrow \sigma'' \models Q'_n,\ \textit{and} \\ \chi = \textit{exc} & \Rightarrow \sigma'' \models Q'_e \end{array}$$

and we prove $\mathcal{A} \models \{\ P\ \}\quad s_1 \quad \{\ Q'_n\ ,\ Q'_e\ \}$
$\square$

Conjunction Rule

We have to prove:

$$\begin{array}{l} \mathcal{A} \vdash \{\ P^1 \wedge P^2\ \}\quad s_1 \quad \{\ Q^1_n \wedge Q^2_n\ ,\ Q^1_e \wedge Q^2_e\ \}\ \ \textit{implies} \\ \mathcal{A} \models \{\ P^1 \wedge P^2\ \}\quad s_1 \quad \{\ Q^1_n \wedge Q^2_n\ ,\ Q^1_e \wedge Q^2_e\ \} \end{array}$$

using the induction hypotheses:

$$\begin{array}{l} \mathcal{A} \vdash \{\ P^1\ \}\quad s_1 \quad \{\ Q^1_n\ ,\ Q^1_e\ \}\ \ \textit{implies}\ \mathcal{A} \models \{\ P^1\ \}\quad s_1 \quad \{\ Q^1_n\ ,\ Q^1_e\ \} \\ \mathcal{A} \vdash \{\ P^2\ \}\quad s_1 \quad \{\ Q^2_n\ ,\ Q^2_e\ \}\ \ \textit{implies}\ \mathcal{A} \models \{\ P^2\ \}\quad s_1 \quad \{\ Q^2_n\ ,\ Q^2_e\ \} \end{array}$$

This holds applying the definition of $\models_N$, and the hypotheses.
$\square$

Disjunction Rule

The proof is similar to the conjunction rule proof.

□

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 poscondition 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:*

$$\{Q_n^{T@m}, Q_e^{T@m}\} \triangleq \mathit{SP}(T@m, \$ = \$')$$

Let $\mathcal{A}_0$ *be the sequent defined as follows:*

$$\mathcal{A}_0 = \bigwedge_{T@m} \{\ \$ = \$'\ \}\quad T@m\quad \{\ Q_n^{T@m}\ ,\ Q_e^{T@m}\ \}$$

$$\textit{If}\quad \models \{\ P\ \}\quad s\quad \{\ Q_n\ ,\ Q_e\ \}\quad \textit{then}\quad \mathcal{A}_0 \vdash\ \{\ P\ \}\quad s\quad \{\ Q_n\ ,\ Q_e\ \}$$

Before proving Lemma 9, we use it to prove the completeness theorem.

Lemma 10 (Sequent T@m)**.**

$$\mathcal{A}, \left\{\ \$ = \$'\ \right\}\quad T@m\quad \left\{\ Q_n^{T@m}\ ,\ Q_e^{T@m}\ \right\} \vdash \left\{\ P\ \right\}\quad s\quad \left\{\ Q_n\ ,\ Q_e\ \right\}\quad \textit{implies}$$

$$\mathcal{A} \vdash \left\{\ P\ \right\}\quad s\quad \left\{\ Q_n\ ,\ Q_e\ \right\}$$

Proof. By induction on the derivation

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

□

Now, we prove the completeness theorem:
Proof of Completeness Theorem We have to prove:

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

Assume $\models \{ \ P \ \} \quad s \quad \{ \ Q_n \ , \ Q_e \ \}$. Then, applying the Lemma 9 we get:

$$\mathcal{A}_0 \vdash \ \{ \ P \ \} \quad s \quad \{ \ Q_n \ , \ Q_e \ \}$$

Then by repeated application of Lemma 10 we obtain:

$$\vdash \ \{ \ P \ \} \quad s \quad \{ \ Q_n \ , \ Q_e \ \}$$

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:

$$\models \left\{ \begin{array}{l} (\mathit{safe}(e) \ \wedge \ P[e/x]) \vee \\ (\neg \mathit{safe}(e) \ \wedge \ Q_e) \end{array} \right\} \quad x \ := \ e \quad \{ \ P \ , \ Q_e \ \} \ \mathit{implies}$$

$$\mathcal{A}_0 \vdash \left\{ \begin{array}{l} (\mathit{safe}(e) \ \wedge \ P[e/x]) \vee \\ (\neg \mathit{safe}(e) \ \wedge \ Q_e) \end{array} \right\} \quad x \ := \ e \quad \{ \ P \ , \ Q_e \ \}$$

Let P' be $(\mathit{safe}(e) \ \wedge \ P[e/x]) \ \vee \ (\neg \mathit{safe}(e) \ \wedge \ Q_e)$.
Assume $\models \ \{ \ P' \ \} \quad x \ := \ e \quad \{ \ Q'_n \ , \ Q'_e \ \}$, then $\models P' \Rightarrow Q'_n[e/x]$. Then by the assignment axiom and the consequence rule we prove:

$$\mathcal{A}_0 \vdash \ \{ \ P' \ \} \quad x \ := \ e \quad \{ \ Q'_n \ , \ Q'_e \ \}$$

□

B.3.2 Compound Rule

We have to prove:

$$\models \{\ P\ \}\ \ s_1; s_2\ \ \{\ R_n\ ,\ R_e\ \}\ \ \textit{implies}\ \ \mathcal{A}_0 \vdash\ \{\ P\ \}\ \ s_1; s_2\ \ \{\ R_n\ ,\ R_e\ \}$$

using the hypotheses

$$\begin{array}{l} \models \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ R_e\ \}\ \ \ \textit{implies}\ \ \mathcal{A}_0 \vdash\ \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ R_e\ \} \\ \textit{and} \\ \models \{\ Q_n\ \}\ \ s_2\ \ \{\ R_n\ ,\ R_e\ \}\ \ \textit{implies}\ \ \mathcal{A}_0 \vdash\ \{\ Q_n\ \}\ \ s_2\ \ \{\ R_n\ ,\ R_e\ \} \end{array}$$

Assume $\models \{\ P\ \}\ \ s_1; s_2\ \ \{\ R_n\ ,\ R_e\ \}$. Then

$$\begin{array}{l} \models \{\ P\ \}\ \ s_1\ \ \{\ T_n\ ,\ T_e\ \}\ \ \ \textit{and} \\ \models \{\ T_n\ \}\ \ s_2\ \ \{\ R'_n\ ,\ R'_e\ \} \end{array}$$

where $\{T_n, T_e\}$ and $\{R'_n, R'_e\}$ are the strongest postconditions defined as follows:

$$\begin{array}{l} \{T_n, T_e\} \triangleq s_1(P) \\ \{R_n, R'_e\} \triangleq s_2(T_n) \end{array}$$

By induction hypotheses we have:

$$\begin{array}{l} \mathcal{A}_0 \vdash\ \{\ P\ \}\ \ s_1\ \ \{\ T_n\ ,\ T_e\ \}\ \ \ \textit{and} \\ \mathcal{A}_0 \vdash\ \{\ T_n\ \}\ \ s_2\ \ \{\ R'_n\ ,\ R'_e\ \} \end{array}$$

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:

$$\begin{array}{l} \mathcal{A}_0 \vdash\ \{\ P\ \}\ \ s_1\ \ \{\ T_n\ ,\ R_e\ \}\ \textit{and} \\ \mathcal{A}_0 \vdash\ \{\ T_n\ \}\ \ s_2\ \ \{\ R_n\ ,\ R_e\ \} \end{array}$$

The conclusion $\mathcal{A}_0 \vdash\ \{\ P\ \}\ \ s_1; s_2\ \ \{\ R_n\ ,\ R_e\ \}$ follows by the compound rule.
□

B.3.3 Conditional Rule

We have to prove:

$$\models \{\ P\ \}\ \ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}\ \ \{\ Q_n\ ,\ Q_e\ \}\ \ \textit{implies}$$

$$\mathcal{A}_0 \vdash\ \{\ P\ \}\ \ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}\ \ \{\ Q_n\ ,\ Q_e\ \}$$

using the hypotheses

$$\begin{array}{l} \models \{\ P\ \wedge\ e\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}\ \ \textit{implies}\ \ \mathcal{A}_0 \vdash\ \{\ P\ \wedge\ e\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \} \\ \textit{and} \\ \models \{\ Q_n\ \}\ \ s_2\ \ \{\ R_n\ ,\ R_e\ \}\qquad \textit{implies}\ \ \mathcal{A}_0 \vdash\ \{\ P\ \wedge\ \neg e\ \}\ \ s_2\ \ \{\ Q_n\ ,\ Q_e\ \} \end{array}$$

Assume $\models \{\ P\ \}\ \ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}\ \ \{\ Q'_n\ ,\ Q'_e\ \}$. Then,

$$\begin{array}{l} \models \{\ P\ \wedge\ e\ \}\ \ s_1\ \ \{\ Q'_n\ ,\ Q'_e\ \} \\ \models \{\ P\ \wedge\ \neg e\ \}\ \ s_2\ \ \{\ Q'_n\ ,\ Q'_e\ \} \end{array}$$

where $\{T_n, T_e\}$ is the strongest postcondition $\{Q'_n, Q'_e\} \triangleq s_1(P\ \wedge\ e) \cup s_2(P\ \wedge\ \neg e)$ Then by induction hypotheses, we know:

$$\begin{array}{l} \mathcal{A}_0 \vdash\ \{\ P\ \wedge\ e\ \}\ \ s_1\ \ \{\ Q'_n\ ,\ Q'_e\ \} \\ \mathcal{A}_0 \vdash\ \{\ P\ \wedge\ \neg e\ \}\ \ s_2\ \ \{\ Q'_n\ ,\ Q'_e\ \} \end{array}$$

Since $Q'_n \Rightarrow Q_n$ and $Q'_e \Rightarrow Q_e$, applying the conditional rule and the rule of consequence we obtain:

$$\mathcal{A}_0 \vdash\ \{\ P\ \}\ \ \texttt{if}\ e\ \texttt{then}\ s_1\ \texttt{else}\ s_2\ \texttt{end}\ \ \{\ R_n\ ,\ R_e\ \}$$

□

B.3.4 Check Axiom

We have to prove:

$$\models\ \{\ P\ \}\ \ \texttt{check}\ e\ \texttt{end}\ \ \{\ (P\ \wedge\ e\ )\ ,\ (P\ \wedge\ \neg e\ )\ \}\ \ \textit{implies}$$

$$\mathcal{A}_0 \vdash\ \{\ P\ \}\ \ \texttt{check}\ e\ \texttt{end}\ \ \{\ (P\ \wedge\ e\ )\ ,\ (P\ \wedge\ \neg e\ )\ \}$$

Assume $\models \{\ P\ \}\ \ \texttt{check}\ e\ \texttt{end}\ \ \{\ Q'_n\ ,\ Q'_e\ \}$. For $Q'_n \triangleq (P \wedge e)$ and $Q'_e \triangleq (P \wedge \neg e)$, the conclusion follows by applying the rule of consequence, and the check axiom.
□

B.3.5 Loop Rule

We have to prove:

$$\begin{array}{l} \models \{\ I\ \}\ \ \texttt{until}\ e\ \texttt{loop}\ s_1\ \texttt{end}\ \ \{\ (I\ \wedge\ e)\ ,\ R_e\ \} \\ \textit{implies} \\ \mathcal{A}_0 \vdash\ \{\ I\ \}\ \ \texttt{until}\ e\ \texttt{loop}\ s_1\ \texttt{end}\ \ \{\ (I\ \wedge\ e)\ ,\ R_e\ \} \end{array}$$

using the hypotheses

$$\models \{\ \neg e\ \wedge\ I\ \}\ \ s_2\ \ \{\ I\ ,\ R_e\ \}\ \ \textit{implies}\ \ \mathcal{A}_0 \vdash\ \{\ \neg e\ \wedge\ I\ \}\ \ s_2\ \ \{\ I\ ,\ R_e\ \}$$

Assume $\models \{\ I\ \}$ `until` e `loop` s_1 `end` $\{\ T_n\ ,\ T_e\ \}$

Let $\{P_n^0, R_e'\}$ be the strongest postcondition $\{P_n^0, P_e^0\} \triangleq s_1(P)$. Let $\{P_n^{i+1}, P_e^{i+1}\}$ be the strongest postcondition $\{P_n^{i+1}, P_e^{i+1}\} \triangleq s_2(P_n^i)$. Let I' be the invariant $I' \triangleq \cup_i P_n^i$ and R_e' be $R_e' \triangleq \cup_i P_e^i$. Then by induction hypotheses we get:

$$\mathcal{A}_0 \vdash\ \{\ \neg e\ \wedge\ I'\ \}\ \ s_2\ \ \{\ I'\ ,\ R_e'\ \}$$

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:

$\mathcal{A}_0 \vdash\ \{\ I\ \}$ `until` e `loop` s_1 `end` $\{\ (I \wedge e)\ ,\ R_e\ \}$

□

B.3.6 Read Attribute Axiom

We have to prove:

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

$$\mathcal{A}_0 \vdash \left\{ \begin{array}{l} (y \neq \textit{Void}\ \wedge\ P[\$(\textit{instvar}(y, S@a))/x]) \vee \\ (y = \textit{Void}\ \wedge\ Q_e) \end{array} \right\}\ \ x := y.S@a\ \ \{\ P\ ,\ Q_e\ \}$$

Let P' be $P' \triangleq (y \neq \textit{Void}\ \wedge\ P[\$(\textit{instvar}(y, S@a))/x])\ \vee\ (y = \textit{Void}\ \wedge\ Q_e)$.

Assume that $\models \{\ P'\ \}\ \ x := y.S@a\ \ \{\ Q_n'\ ,\ Q_e'\ \}$ holds, then $\models P' \Rightarrow Q_n'[(\textit{instvar}(y, S@a))/x]$. Then, by the read attribute axiom and the consequence rule, we prove:

$$\mathcal{A}_0 \vdash\ \{\ P'\ \}\ \ x := y.S@a\ \ \{\ Q_n'\ ,\ Q_e'\ \}$$

□

B.3.7 Write Attribute Axiom

We have to prove:

$$\models \left\{ \begin{array}{l} (y \neq \mathit{Void} \ \wedge \ P[\$\langle \mathit{instvar}(y, S@a) := e\rangle/\$]) \vee \\ (y = \mathit{Void} \ \wedge \ Q_e) \end{array} \right\} \ y.S@a := e \ \ \{\ P\ ,\ Q_e\ \}$$

implies

$$\mathcal{A}_0 \vdash \left\{ \begin{array}{l} (y \neq \mathit{Void} \ \wedge \ P[\$\langle \mathit{instvar}(y, S@a) := e\rangle/\$]) \vee \\ (y = \mathit{Void} \ \wedge \ Q_e) \end{array} \right\} \ y.S@a := e \ \ \{\ P\ ,\ Q_e\ \}$$

Let P' be $P' \triangleq (y \neq \mathit{Void} \ \wedge \ P[\$\langle \mathit{instvar}(y, S@a) := e\rangle/\$]) \ \vee \ (y = \mathit{Void} \ \wedge \ Q_e)$.
Assume $\models \{\ P'\ \} \ \ y.S@a := e \ \ \{\ Q'_n\ ,\ Q'_e\ \}$ *holds*, then
$\models P' \Rightarrow Q'_n[\$\langle \mathit{instvar}(y, S@a) := e\rangle/\$]$. Then, by the write attribute axiom and the consequence rule, we prove:

$$\mathcal{A}_0 \vdash \ \{\ P'\ \} \ \ y.S@a := e \ \ \{\ Q'_n\ ,\ Q'_e\ \}$$

□

B.3.8 Local Rule

We have to prove:

$$\begin{array}{l} \models \{\ P\ \} \ \ \texttt{local} \ \ v_1 : T_1; \ ... \ v_n : T_n; \ s \ \ \{\ Q_n\ ,\ Q_e\ \} \ \ \mathit{implies} \\ \mathcal{A}_0 \vdash \ \{\ P\ \} \ \ \texttt{local} \ \ v_1 : T_1; \ ... \ v_n : T_n; \ s \ \ \{\ Q_n\ ,\ Q_e\ \} \end{array}$$

using the induction hypothesis:

$$\begin{array}{l} \models \{\ P \ \wedge \ v_1 = \mathit{init}(T_1) \ \wedge \ ... \wedge \ v_n = \mathit{init}(T_n)\ \} \ \ s \ \ \{\ Q_n\ ,\ Q_e\ \} \qquad \mathit{implies} \\ \mathcal{A}_0 \vdash \ \{\ P \ \wedge \ v_1 = \mathit{init}(T_1) \ \wedge \ ... \wedge \ v_n = \mathit{init}(T_n)\ \} \ \ s \ \ \{\ Q_n\ ,\ Q_e\ \} \end{array}$$

Assume $\models \{\ P\ \} \ \ \texttt{local} \ \ v_1 : T_1; \ ... \ v_n : T_n; \ s \ \ \{\ Q_n\ ,\ Q_e\ \}$, then

$$\models \{\ P \ \wedge \ v_1 = \mathit{init}(T_1) \ \wedge \ ... \wedge \ v_n = \mathit{init}(T_n)\ \} \ \ s \ \ \{\ Q'_n\ ,\ Q'_e\ \}$$

where $\{Q'_n, Q'_e\}$ is the strongest postcondition:

$$\{Q'_n, Q'_e\} \triangleq s(P \ \wedge \ v_1 = \mathit{init}(T_1) \ \wedge \ ... \wedge \ v_n = \mathit{init}(T_n))$$

By induction hypothesis we have:

$$\mathcal{A}_0 \vdash \ \{\ P \ \wedge \ v_1 = \mathit{init}(T_1) \ \wedge \ ... \wedge \ v_n = \mathit{init}(T_n)\ \} \ \ s \ \ \{\ Q'_n\ ,\ Q'_e\ \}$$

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:

$$\mathcal{A}_0 \vdash \; \{ \; P \; \} \;\; \texttt{local} \;\; v_1 : T_1; \; ... \; v_n : T_n; \; s \;\; \{ \; Q_n \; , \; Q_e \; \}$$

□

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 `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 $Retry = true$ then the postcondition Q_n^i implies P_n^i. If $Retry = false$ then the postcondition Q_n^i implies $Q_n^i \; \wedge \; \neg Retry \; \wedge \; Q_e^i$. Furthermore, Q_e^i implies $Q_n^i \; \wedge \; \neg Retry \; \wedge \; Q_e^i$

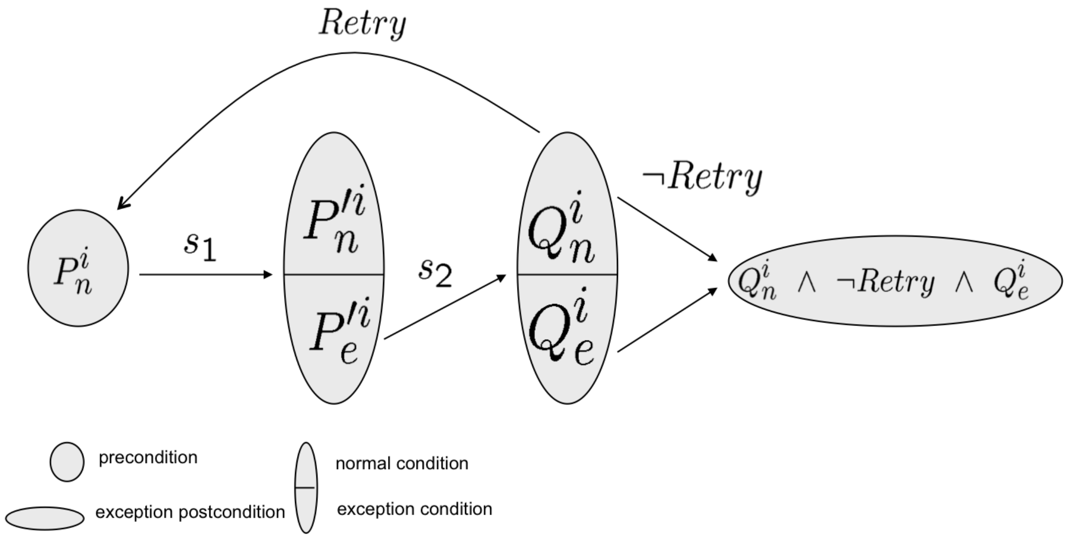

Fig. B.1: Completeness proof

We have to prove:

$$\begin{array}{l} \models \{ \; P \; \} \;\; s_1 \; \texttt{rescue} \; s_2 \;\; \{ \; Q_n \; , \; R_e \; \} \\ \textit{implies} \\ \mathcal{A}_0 \vdash \; \{ \; P \; \} \;\; s_1 \; \texttt{rescue} \; s_2 \;\; \{ \; Q_n \; , \; R_e \; \} \end{array}$$

using the hypotheses

$\models \{\ I_r\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}$ *implies* $\mathcal{A}_0 \vdash\ \{\ I_r\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}$ *and*

$\models \{\ Q_e\ \}\ \ s_2\ \ \{\ Retry \Rightarrow I_r\ \wedge\ \neg Retry \Rightarrow R_e\ ,\ R_e\ \}$
implies
$\mathcal{A}_0 \vdash\ \{\ Q_e\ \}\ \ s_2\ \ \{\ Retry \Rightarrow I_r\ \wedge\ \neg Retry \Rightarrow R_e\ ,\ R_e\ \}$ *and*

$P\ \Rightarrow\ I_r$

Assume $\models \{\ P\ \}\ \ s_1$ `rescue` $s_2\ \ \{\ Q_n\ ,\ Q_e\ \}$. Let

$$P_n^0 \triangleq P \tag{B.21}$$
$$\{P'^i_n, P'^i_e\} \triangleq s_1(P_n^i) \tag{B.22}$$
$$\{Q_n^i, Q_e^i\} \triangleq s_2(P'^i_e) \tag{B.23}$$
$$P_n^{i+1} \triangleq Q_n^i\ \wedge\ Retry \tag{B.24}$$
$$I_r \triangleq \cup_i P_n^i \tag{B.25}$$
$$T_n \triangleq \cup_i P'^i_n \tag{B.26}$$
$$T_e \triangleq \cup_i P'^i_e \tag{B.27}$$
$$R_e \triangleq \cup_i((Q_n^i\ \wedge\ \neg Retry)\ \vee Q_e^i) \tag{B.28}$$

We have $T_n \Rightarrow Q_n$ and $R_e \Rightarrow Q_e$. Then,

$$\begin{array}{l} \models \{\ P_n^i\ \}\ \ s_1\ \ \{\ P'^i_n\ ,\ P'^i_e\ \},\ \textit{and} \\ \models \{\ P'^i_e\ \}\ \ s_2\ \ \{\ Q_n^i\ ,\ Q_e^i\ \},\ \textit{for all } i \end{array}$$

Therefore,

$$\begin{array}{l} \models \{\ \cup_i\ P_n^i\ \}\ \ s_1\ \ \{\ \cup_i\ P'^i_n\ ,\ \cup_i\ P'^i_e\ \}\ \textit{and} \\ \models \{\ \cup_i\ P'^i_e\ \}\ \ s_2\ \ \{\ \cup_i\ Q_n^i\ ,\ \cup_i\ Q_e^i\ \} \end{array}$$

Then by induction hypotheses

$$\begin{array}{l} \mathcal{A}_0 \vdash\ \{\ \cup_i\ P_n^i\ \}\ \ s_1\ \ \{\ \cup_i\ P'^i_n\ ,\ \cup_i\ P'^i_e\ \}\ \textit{and} \\ \mathcal{A}_0 \vdash\ \{\ \cup_i\ P'^i_e\ \}\ \ s_2\ \ \{\ \cup_i\ Q_n^i\ ,\ \cup_i\ Q_e^i\ \} \end{array}$$

Since $\cup_i\ P_n^i \Rightarrow P$, and $\cup_i\ P'^i_n \Rightarrow Q_n$, and $\cup_i\ Q_e^i \Rightarrow Q_e$, and the rule of consequence, we get

$$\mathcal{A}_0 \vdash\ \{\ I_r\ \}\ \ s_1\ \ \{\ Q_n\ ,\ T_e\ \} \tag{B.29}$$

By (B.24) we know $Q_n^i \Rightarrow (\mathit{Retry} \Rightarrow P_n^i)$ and by (B.28) $Q_n^i \Rightarrow ((\neg \mathit{Retry}) \Rightarrow R_e)$. Then, since $I_r = \cup_i P_n^i$ we get $Q_n^i \Rightarrow (\mathit{Retry} \Rightarrow I_r)$ and since $R_e \Rightarrow Q_e$ we get $Q_n^i \Rightarrow ((\neg \mathit{Retry}) \Rightarrow Q_e)$. Then $\cup_i\ Q_n^i \Rightarrow (\mathit{Retry} \Rightarrow I_r \wedge \neg \mathit{Retry} \Rightarrow Q_e)$.

Then, by (B.28) $\cup_i\ Q_e^i \Rightarrow R_e$, $R_e \Rightarrow Q_e$, and the rule of consequence, we prove:

$$\mathcal{A}_0 \vdash \left\{\ T_e\ \right\}\ \ s_2\ \ \left\{\ (\mathit{Retry} \Rightarrow I_r \wedge \neg \mathit{Retry} \Rightarrow Q_e\ ,\ Q_e\ \right\} \tag{B.30}$$

To finish the proof, we need to prove $P \Rightarrow I_r$. This holds because $P = P_n^0$ and $I_r = \cup_i P_n^i$. Then from B.29 and B.30, and applying the `rescue` rule we get:

$$\mathcal{A}_0 \vdash \left\{\ P\ \right\}\ \ s_1\ \texttt{rescue}\ s_2\ \ \left\{\ Q_n\ ,\ Q_e\ \right\}$$

□

B.3.10 Routine Implementation Rule

We have to prove:

$$\models \left\{\ P\ \right\}\ \ T@m\ \ \left\{\ Q_n\ ,\ Q_e\ \right\}\ \ \textit{implies}\ \ \mathcal{A}_0 \vdash \left\{\ P\ \right\}\ \ T@m\ \ \left\{\ Q_n\ ,\ Q_e\ \right\}$$

Assume $\models \left\{\ P\ \right\}\ \ T@m\ \ \left\{\ Q_n\ ,\ Q_e\ \right\}$.

$$\dfrac{\dfrac{\dfrac{\mathcal{A}_0 \vdash \left\{\ \$ = \$'\ \right\}\ \ T@m\ \ \left\{\ Q_n^{T@m}\ ,\ Q_e^{T@m}\ \right\}}{\mathcal{A}_0 \vdash \left\{\ \$ = \$' \wedge P[\$'/\$]\ \right\}\ \ T@m\ \ \left\{\ Q_n^{T@m} \wedge P[\$'/\$]\ ,\ Q_e^{T@m} \wedge P[\$'/\$]\ \right\}}}{\mathcal{A}_0 \vdash \left\{\ \$ = \$' \wedge P[\$'/\$]\ \right\}\ \ T@m\ \ \left\{\ Q_n\ ,\ Q_e\ \right\}}\ \text{B.31,B.32}}{\mathcal{A}_0 \vdash \left\{\ P\ \right\}\ \ T@m\ \ \left\{\ Q_n\ ,\ Q_e\ \right\}}\ \exists \$'$$

where (B.31), (B.32) are the following implications:

$$Q_n^{T@m}\ \wedge\ P[\$'/\$]\ \ \Rightarrow\ \ Q_n \tag{B.31}$$
$$Q_e^{T@m}\ \wedge\ P[\$'/\$]\ \ \Rightarrow\ \ Q_e \tag{B.32}$$

B.3.11 Routine Invocation Rule

We have to prove:

$$\begin{array}{l} \models \left\{\ P\ \right\}\ \ x := y.T{:}m(e)\ \ \left\{\ Q_n\ ,\ Q_e\ \right\} \\ \quad \textit{implies} \\ \mathcal{A}_0 \vdash \left\{\ P\ \right\}\ \ x := y.T{:}m(e)\ \ \left\{\ Q_n\ ,\ Q_e\ \right\} \end{array}$$

Assume $\models \left\{\ P\ \right\}\ \ x := y.T{:}m(e)\ \ \left\{\ Q_n\ ,\ Q_e\ \right\}$. Then, by definition of $\models$ we obtain:

$$\models \{\ P[Result'/Result]\ \}\ \ Result := y.T{:}m(e)\ \ \{\ Q_n[Result'/Result, Result/x]\ ,\ Q_e[Result'/Result]\ \}$$

Let P', Q'_n and Q'_e be the following pre and postconditions:

$$P' \triangleq P \begin{bmatrix} Result'/Result, & Current'/Current, \\ Current/y & \end{bmatrix} \wedge\ p = e$$

$$Q'_n \triangleq Q_n \begin{bmatrix} Result'/Result, & Result/x \\ Current'/Current, & Current/y \end{bmatrix}$$

$$Q'_e \triangleq Q_e \begin{bmatrix} Result'/Result, & Current'/Current, \\ Current/y & \end{bmatrix}$$

By definition of $\models$, we get:

$$\models \{\ P'\ \}\ \ Result := Current.T{:}m(p)\ \ \{\ Q'_n\ ,\ Q'_e\ \}$$

Then, by definition of $\models$, we obtain:

$$\models \{\ P'\ \}\ \ T{:}m(p)\ \ \{\ Q'_n\ ,\ Q'_e\ \}$$

Then, we obtain the following derivation:

$$\frac{\mathcal{A}_0 \vdash\ \{\ P'\ \}\ \ T{:}m(p)\ \ \{\ Q'_n\ ,\ Q'_e\ \}}{\mathcal{A}_0 \vdash\ \{\ P''\ \}\ \ x := y.T{:}m(e)\ \ \{\ Q''_n\ ,\ Q''_e\ \}}\ \text{invocation rule}$$

where P'', Q''_n, and Q''_e are defined as follows:

$$P'' \triangleq \begin{array}{l} y \neq Void\ \wedge\ P'[y/Current, e/p] \\ y = Void \wedge\ Q'_e[y/Current] \end{array}$$

$$Q''_n \triangleq Q'_n[y/Current, x/Result]$$

$$Q''_e \triangleq Q'_e[y/Current]$$

Unfolding the definition of P', and Q'_e we obtain

$$P'' \triangleq \left(\begin{array}{l} y \neq \mathit{Void} \ \wedge \ P \left[\begin{array}{l} \mathit{Result'}/\mathit{Result}, \mathit{Current'}/\mathit{Current}, \\ \mathit{Current}/y, \ y/\mathit{Current}, \end{array} \right] \wedge \ e = e \ \wedge \\ \\ y = \mathit{Void} \wedge \ Q'_e \left[\begin{array}{l} \mathit{Current}/y, y/\mathit{Current}, \\ \mathit{Result'}/\mathit{Result}, \ \mathit{Current'}/\mathit{Current} \end{array} \right] \end{array} \right)$$

$$\equiv \left(\begin{array}{l} y \neq \mathit{Void} \ \wedge \ P \left[\begin{array}{l} \mathit{Result'}/\mathit{Result} \\ \mathit{Current'}/\mathit{Current} \end{array} \right] \\ \\ y = \mathit{Void} \wedge \ Q'_e \left[\begin{array}{l} \mathit{Result'}/\mathit{Result} \\ \mathit{Current'}/\mathit{Current} \end{array} \right] \end{array} \right)$$

Also, unfolding the definition of Q'_n and Q'_e we know:

$$Q''_n \triangleq Q_n \left[\begin{array}{l} \mathit{Result'}/\mathit{Result}, \mathit{Current'}/\mathit{Current}, \\ \mathit{Current}/y, \ y/\mathit{Current}, \\ \mathit{Result}/x, x/\mathit{Result} \end{array} \right]$$

$$\equiv Q_n \left[\ \mathit{Result'}/\mathit{Result}, \mathit{Current'}/\mathit{Current} \ \right]$$

$$Q''_e \triangleq Q_e \left[\begin{array}{l} \mathit{Result'}/\mathit{Result}, \mathit{Current'}/\mathit{Current}, \\ \mathit{Current}/y, \ y/\mathit{Current} \end{array} \right]$$

$$\equiv Q_e \left[\ \mathit{Result'}/\mathit{Result}, \mathit{Current'}/\mathit{Current} \ \right]$$

Thus, the only replacement used in P'', Q''_n, and Q''_e is $\mathit{Result'}/\mathit{Result}$ and $\mathit{Current'}/\mathit{Current}$. Now applying the *invoc_var_rule* with $\mathit{Result'}$ and $\mathit{Current'}$ we obtain the following derivation:

$$\frac{\mathcal{A}_0 \vdash \{ \ P'' \ \} \ \ x := y.T{:}m(p) \ \ \{ \ Q''_n \ , \ Q''_e \ \}}{\mathcal{A}_0 \vdash \{ \ \ (y \neq \mathit{Void} \ \wedge \ P) \vee \ y = \mathit{Void} \wedge \ Q_e \ \ \} \ \ x := y.T{:}m(p) \ \ \{ \ Q_n \ , \ Q_e \ \}} \ \text{invoc\_var\_rule}$$

Finally, since we know $(P \wedge \ y = \mathit{Void}) \Rightarrow Q_e$ from the hypothesis, applying the rule of consequence we prove:

$$\mathcal{A}_0 \vdash \{ \ P \ \} \ \ x := y.T{:}m(e) \ \ \{ \ Q_n \ , \ Q_e \ \}$$

□

B.3.12 Virtual Routines

To prove this case, $T{:}m$, we use the following lemma:

Lemma 11 (Subtypes)**.**

$$\forall T' \preceq T : \ \models \left\{ \ P \ \wedge \ \tau(\mathit{Current}) = T' \ \right\} \quad T{:}m \quad \left\{ \ Q_n \ , \ Q_e \ \right\} \quad \mathit{then}$$

$$\vdash \ \left\{ \ P \ \wedge \ \tau(\mathit{Current}) = T' \ \right\} \quad T'{:}\,m \quad \left\{ \ Q_n \ , \ Q_e \ \right\}$$

Proof. The proof follows from the proof rule for

$$\vdash \ \left\{ \ P \ \wedge \ \tau(\mathit{Current}) = T' \ \right\} \quad T'{:}\,m \quad \left\{ \ Q_n \ , \ Q_e \ \right\}$$

and completeness for the routine implementation rule presented in Section B.3.10.

□

Now, we prove the case $T{:}m$ as follows. We know:

$$\models \left\{ \ P \ \right\} \quad T{:}m \quad \left\{ \ Q_n \ , \ Q_e \ \right\}$$

Applying Lemma 11 to all descendants of T, and applying the subtype rule and the consequence rule get:

$$\vdash \ \left\{ \ P \ \right\} \quad T{:}m \quad \left\{ \ Q_n \ , \ Q_e \ \right\}$$

□

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)

$$\frac{Tree_1}{\mathcal{A} \vdash \left\{ \; P \; \right\} \quad m \quad \left\{ \; Q_n \; , \; Q_e \; \right\}} \qquad then$$

$$\vdash \; \nabla_B \left(\frac{Tree_1}{\mathcal{A} \vdash \left\{ \; P \; \right\} \quad m \quad \left\{ \; Q_n \; , \; Q_e \; \right\}} \right)$$

Theorem 4 (Section 6.7)

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

Theorem 5 (Section 7.4)

$$
\begin{array}{l}
\forall b : BoolExp, \;\; t : TypeFunc, \;\; e : Expr, \;\; c : CallRoutine, \;\; p : Argument : \\
(wellF_C \;\; b) \Rightarrow (value_C \;\; b \;\; h_1 \;\; h_2 \;\; s) = ((\nabla_C \;\; b) \;\; h_1 \;\; h_2 \;\; s) \quad and \\
(wellF_T \;\; t) \Rightarrow (value_T \;\; t \;\; h_1 \;\; h_2 \;\; s) = ((\nabla_T \;\; t) \;\; h_1 \;\; h_2 \;\; s) \quad and \\
(wellF_{Exp} \;\; e) \Rightarrow (value_{Exp} \;\; e \;\; h_1 \;\; h_2 \;\; s) = ((\nabla_{Exp} \;\; e) \;\; h_1 \;\; h_2 \;\; s) \quad and \\
(wellF_{Call} \;\; c) \Rightarrow (value_{Call} \;\; c \;\; h_1 \;\; h_2 \;\; s) = ((\nabla_{Call} \;\; c) \;\; h_1 \;\; h_2 \;\; s) \quad and \\
(wellF_{Arg} \;\; p) \Rightarrow (value_{Arg} \;\; p \;\; h_1 \;\; h_2 \;\; s) = ((\nabla_{Arg} \;\; p) \;\; h_1 \;\; h_2 \;\; s)
\end{array}
$$

The proofs of Theorem 3 and Theorem 4 run by induction on the structure of the derivation tree for $\{P\} \; s_1 \; \{Q_n, Q_e\}$. These proofs are presented in Section C.2 and Section C.4. 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-indepent 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_{imp} and T_{tm} be the following proof trees:

$$T_{imp} \equiv \frac{Tree_1}{\mathcal{A} \vdash \{\ \tau(Current) = T\ \wedge\ P\ \}\quad imp(T, m)\quad \{\ Q_n\ ,\ Q_e\ \}}$$

$$T_{tm} \equiv \frac{Tree_2}{\mathcal{A} \vdash \{\ \tau(Current) \prec T\ \wedge\ P\ \}\qquad T{:}m\qquad \{\ Q_n\ ,\ Q_e\ \}}$$

where $Tree_1$ and $Tree_2$ are the derivations used to prove the Hoare triples of $imp(T, m)$ and $T{:}m$ respectively.

We have to prove:

$$\frac{T_{imp}\qquad T_{tm}}{\mathcal{A} \vdash \left\{\ \tau(Current) \preceq T\ \wedge\ P\ \right\}\qquad T{:}m\qquad \left\{\ Q_n\ ,\ Q_e\ \right\}}$$

implies

$$\vdash \frac{\nabla_B(\ T_{imp}\ )\qquad \nabla_B(\ T_{tm}\ )}{\mathcal{A} \vdash \left\{\ \tau(Current) \preceq T\ \wedge\ P\ \right\}\qquad T{:}m\qquad \left\{\ Q_n\ ,\ Q_e\ \right\}}\ \text{cil class rule}$$

using the induction hypotheses:

$$T_{imp}\quad implies\quad \vdash \nabla_B(\ T_{imp})$$
$$T_{tm}\quad implies\quad \vdash \nabla_B(\ T_{tm})$$

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:

$$\vdash \nabla_B(\ T_{imp})$$
$$\vdash \nabla_B(\ T_{tm})$$

Applying the *class rule* in the CIL logic using the above CIL proofs, we conclude:

$$\vdash \frac{\nabla_B(\ T_{imp}\ )\qquad \nabla_B(\ T_{tm}\ )}{\mathcal{A} \vdash \left\{\ \tau(Current) \preceq T\ \wedge\ P\ \right\}\qquad T{:}m\qquad \left\{\ Q_n\ ,\ Q_e\ \right\}}\ \text{cil class rule}$$

□

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:

$$T_{S_1} \equiv \frac{\mathit{Tree}_1}{\mathcal{A} \vdash \{\ P'\ \}\ \ s_1\ \ \{\ Q'_n\ ,\ Q'_e\ \}} \qquad T_{S_2} \equiv \frac{\mathit{Tree}_2}{\mathcal{A} \vdash \{\ P''\ \}\ \ s_1\ \ \{\ Q''_n\ ,\ Q''_e\ \}}$$

We have to prove:

$$\frac{T_{S_1} \qquad T_{S_2}}{\mathcal{A} \vdash \{\ P' \wedge P''\ \}\ \ s_1\ \ \{\ Q'_n \wedge Q''_n\ ,\ Q'_e \wedge Q''_e\ \}}$$

implies

$$\frac{\nabla_B(\ T_{S_1}) \qquad \nabla_B(\ T_{S_2})}{\mathcal{A} \vdash \{\ P' \wedge P''\ \}\ \ s_1\ \ \{\ Q'_n \wedge Q''_n\ ,\ Q'_e \wedge Q''_e\ \}}\ \text{cil conjunction rule}$$

using the induction hypotheses:

$$\begin{array}{l} T_{S_1}\ \ \mathit{implies}\ \ \nabla_B(\ T_{S_1}) \\ T_{S_2}\ \ \mathit{implies}\ \ \nabla_B(\ T_{S_2}) \end{array}$$

Since the application of the *conjunction rule* in the source logic is valid, then applying the induction hypotheses we get:

$$\begin{array}{l} \nabla_B(\ T_{S_1}) \\ \nabla_B(\ T_{S_2}) \end{array}$$

Using the *conjunction rule* in the CIL logic, we show:

$$\frac{\nabla_B(\ T_{S_1}) \qquad \nabla_B(\ T_{S_2})}{\mathcal{A} \vdash \{\ P' \wedge P''\ \}\ \ s_1\ \ \{\ Q'_n \wedge Q''_n\ ,\ Q'_e \wedge Q''_e\ \}}\ \text{cil conjunction rule}$$

□

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:

Lemma 12 (Soundness of Expression Translator)**.**

$$
\begin{array}{l}
(I_{l_{start}} ... I_{l_{end}}) = \nabla_E \, (Q \ \wedge \ \mathit{unshift}(P[e/s(0)]), \ \ e, \ \ \mathit{shift}(Q) \ \wedge \ P, \ l_{start}) \ \wedge \\
(\mathit{shift}(Q) \ \wedge \ P \ \Rightarrow \ \ E_{l_{end+1}}) \\
\Rightarrow \\
\forall \ l \ \in \ l_{start} \ ... \ l_{end} : \vdash \{E_l\} \ I_l
\end{array}
$$

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:

$$
\begin{array}{l}
I_{l_{start}} = \nabla_E \, (Q \ \wedge \ \mathit{unshift}(P[c/s(0)]), \ \ c, \ \ \mathit{shift}(Q) \ \wedge \ P, \ l_{start}) \ \wedge \\
(\mathit{shift}(Q) \ \wedge \ P \ \Rightarrow \ \ E_{l_{start+1}}) \\
\Rightarrow \\
\vdash \{Q \ \wedge \ \mathit{unshift}(P[c/s(0)])\} \ \ l_{start} : \ \mathsf{ldc} \ c
\end{array}
$$

To prove that this bytecode specification is valid, we have to show that the precondition $\{Q \ \wedge \ \mathit{unshift}(P[c/s(0)])\}$ implies the weakest precondition of ldc c. To show this implication, first we prove:

$$Q \ \wedge \ \mathit{unshift}(P[c/s(0)]) \ \ \mathit{implies} \ \ \mathit{wp}(\mathsf{ldc} \ c, \mathit{shift}(Q) \wedge P)$$

then, by the hypothesis we know $(\mathit{shift}(Q) \ \wedge \ P \ \Rightarrow \ \ E_{l_{start+1}})$, and we prove

$$Q \ \wedge \ \mathit{unshift}(P[c/s(0)]) \ \ \mathit{implies} \ \ \mathit{wp}(\mathsf{ldc} \ c, \ E_{l_{start+1}})$$

The implication holds as follows:

$$
\begin{array}{l}
Q \ \wedge \ \mathit{unshift}(P[c/s(0)]) \ \ \mathit{implies} \ \ \mathit{wp}(\mathsf{ldc} \ c, \mathit{shift}(Q) \wedge P) \\
\quad [\mathit{definition \ of \ wp}] \\
Q \ \wedge \ \mathit{unshift}(P[c/s(0)]) \ \ \mathit{implies} \ \ \mathit{unshift}(\ \mathit{shift}(Q) \wedge P[c/s(0)] \ ) \\
\quad [\mathit{definition \ of \ unshift}] \\
Q \ \wedge \ \mathit{unshift}(P[c/s(0)]) \ \ \mathit{implies} \ \ Q \ \wedge \ \mathit{unshift}(P[c/s(0)]) \\
\square
\end{array}
$$

C.3.2 Variables

The translation of variables is described in Section 6.3.2 on page 116. We have to show:

$$
\begin{array}{l}
I_{l_{start}} = \nabla_E\,(Q \;\wedge\; \mathit{unshift}(P[x/s(0)]),\;\; x,\;\; \mathit{shift}(Q) \;\wedge\; P,\; l_{start}) \;\wedge \\
(\mathit{shift}(Q) \;\wedge\; P \;\Rightarrow\;\; E_{l_{start+1}}) \\
\Rightarrow \\
\vdash \{Q \;\wedge\; \mathit{unshift}(P[x/s(0)])\}\;\; l_{start} : \; \mathsf{ldc}\ x
\end{array}
$$

To show that the bytecode specification for $\mathsf{ldc}\ x$ is valid, we prove:

$$Q \;\wedge\; \mathit{unshift}(P[x/s(0)]) \;\; \mathit{implies} \;\; \mathit{wp}(\mathsf{ldc}\ x, \mathit{shift}(Q) \wedge P)$$

then, by the hypothesis we have $\mathit{shift}(Q) \;\wedge\; P \;\Rightarrow\;\; E_{l_{start+1}}$.

The proof is as follows:

$$
\begin{array}{l}
Q \;\wedge\; \mathit{unshift}(P[x/s(0)]) \;\; \mathit{implies} \;\; \mathit{wp}(\mathsf{ldc}\ x, \mathit{shift}(Q) \wedge P) \\
\quad [\mathit{definition\ of\ wp}] \\
Q \;\wedge\; \mathit{unshift}(P[x/s(0)]) \;\; \mathit{implies} \;\; \mathit{unshift}(\;\mathit{shift}(Q) \wedge P[x/s(0)]\;) \\
\quad [\mathit{definition\ of\ unshift}] \\
Q \;\wedge\; \mathit{unshift}(P[x/s(0)]) \;\; \mathit{implies} \;\; Q \;\wedge\; \mathit{unshift}(P[x/s(0)]) \\
\Box
\end{array}
$$

C.3.3 Binary Expressions

Binary expressions has been translated in Section 6.3.3 on page 116. We have to prove:

$$
\begin{array}{l}
I_{l_{start}...l_c} = \nabla_E\,(Q \;\wedge\; \mathit{unshift}(P[e_1\ op\ e_2/s(0)]),\;\; e_1\ op\ e_2,\;\; \mathit{shift}(Q) \;\wedge\; P,\; l_{start}) \;\wedge \\
(\mathit{shift}(Q) \;\wedge\; P \;\Rightarrow\;\; E_{l_{c+1}}) \\
\Rightarrow \\
\forall\; l \in l_{start}...l_c \vdash \{E_l\} : \;\; I_l
\end{array}
$$

using the hypotheses:

$l1_{start}...l1_{end} =$
$\nabla_E(\ Q\ \wedge\ \mathit{unshift}(P[e_1\ op\ e_2/s(0)])\ ,\ \ e_1\ \ ,\ \mathit{shift}(Q)\ \wedge\ P[s(0)\ op\ e_2/s(0)]\ ,\ l1_{start})\ \wedge$
$(\mathit{shift}(Q)\ \wedge\ P[s(0)\ op\ e_2/s(0)]\ \Rightarrow\ \ E_{l1_{end}})$
$\Rightarrow$
$\forall\ l \in l1_{start}...l1_{end} \vdash \{E_l\}:\ I_l$

and
$l2_{start}...l2_{end} =$
$\nabla_E(\ \mathit{shift}(Q)\ \wedge\ P[s(0)\ op\ e_2/s(0)]\ ,\ \ e_2\ \ ,\ \mathit{shift}^2(Q)\ \wedge\ \mathit{shift}\ P[s(1)\ op\ s(0)/s(1)]\ ,\ l2_{start})$
$\wedge\ (\mathit{shift}(Q)\ \wedge\ P[s(0)\ op\ e_2/s(0)]\ \Rightarrow\ \ E_{l2_{end}})$
$\Rightarrow$
$\forall\ l \in l2_{start}...l2_{end} \vdash \{E_l\}:\ I_l$

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 *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:

$I_{l_{start}}...l_b = \nabla_E\,(Q\ \wedge\ \mathit{unshift}(P[unop\ e/s(0)]),\ \ unop\ e,\ \ \mathit{shift}(Q)\ \wedge\ P,\ l_{start})\ \wedge$
$(\mathit{shift}(Q)\ \wedge\ P\ \Rightarrow\ \ E_{l_{b+1}})$
$\Rightarrow$
$\forall\ l \in l_{start}...l_b \vdash \{E_l\}:\ I_l$

using the hypothesis:

$l1_{start}...l1_{end} =$
$\nabla_E(\ Q\ \wedge\ \mathit{unshift}(P[unop\ e/s(0)])\ ,\ \ e\ \ ,\ \mathit{shift}(Q)\ \wedge\ P[unop\ s(0)/s(0)]\ ,\ l1_{start})\ \wedge$
$(\mathit{shift}(Q)\ \wedge\ P[unop\ s(0)/s(0)]\ \ \Rightarrow\ \ E_{l1_{end}})$
$\Rightarrow$
$\forall\ l \in l1_{start}...l1_{end} \vdash \{E_l\}:\ I_l$

To apply the induction hypothesis, we need to show:

$$(\mathit{shift}(Q) \ \wedge \ P[\mathit{unop}\ s(0)/s(0)] \ \ \Rightarrow \ \ E_{l1_{end}})$$

Since $E_{l1_{end}}$ is equal to E_{l_b}, and E_{l_b} is defined as:

$$\mathit{shift}(Q) \ \wedge \ P[\mathit{unop}\ s(0)/s(0)]$$

then, the implication holds, and by induction hypothesis we get:

$$\forall\ l \ \in \ l1_{start}...l1_{end} : \vdash \{E_l\}\ I_l$$

Finally, we need to show that the instruction specification at l_b holds. By definition of wp, we need to show:

$$(\mathit{shift}(Q) \ \wedge \ P[\mathit{unop}\ s(0)/s(0)] \ \ \mathit{implies} \ \ wp(\mathit{unop}_{op},\ E_{l_{b+1}})$$

By hypothesis $(\mathit{shift}(Q) \ \wedge \ P \ \Rightarrow \ \ E_{l_{b+1}})$, applying the definition of wp, we prove that

$$(\mathit{shift}(Q) \ \wedge \ P[\mathit{unop}\ s(0)/s(0)] \ \ \mathit{implies} \ \ wp(\mathit{unop}_{op},\ (\mathit{shift}(Q) \ \wedge \ P)$$

□

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 $\mathcal{A} \vdash \{\ P\ \}\ \ s\ \ \{\ Q_n\ ,\ Q_e\ \}$. 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:

$$P' \equiv (\mathit{safe}(e) \ \wedge \ P[e/x]) \ \vee \ (\neg \mathit{safe}(e) \ \wedge \ Q_e)$$

We have to prove:

$$\begin{array}{l}
\vdash \dfrac{}{\mathcal{A} \vdash \left\{\ P'\ \right\}\ \ x := e\ \ \left\{\ P\ ,\ Q_e\ \right\}} \quad \wedge \\
(I_{l_a} ... I_{l_b}) = \nabla_S \left(\dfrac{}{\mathcal{A} \vdash \left\{\ P'\ \right\}\ \ x := e\ \ \left\{\ P\ ,\ Q_e\ \right\}},\ l_a, l_{b+1}, l_{exc} \right) \ \wedge \\
\left(P \ \ \Rightarrow \ \ E_{l_{b+1}}\right) \quad \wedge \\
(\ (Q_e \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \ \ \Rightarrow \ \ E_{l_{exc}}) \quad \wedge \\
\Rightarrow \\
\forall\ l \ \in \ l_a \ ... \ l_b : \vdash \{E_l\}\ I_l
\end{array}$$

Applying Lemma 12 we show that the translation of the expression e produces a valid bytecode proof. The bytecode specification for stloc x can be proved as follows:

$$\begin{array}{l}
\mathit{shift}(\mathit{safe}(e) \ \wedge \ P[e/x]) \wedge \ s(0) = e \ \ \mathit{implies} \ \ \mathit{wp}(\mathsf{stloc}\ x, P) \\
\quad [\mathit{definition\ \ of\ \ wp}] \\
\mathit{shift}(\mathit{safe}(e) \ \wedge \ P[e/x]) \wedge \ s(0) = e \ \ \mathit{implies} \ \ \mathit{shift}(P[s(0)/x]\ )
\end{array}$$

□

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:

$$T_{S_1} \equiv \frac{\mathit{Tree}_1}{\{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ R_e\ \}} \qquad T_{S_2} \equiv \frac{\mathit{Tree}_2}{\{\ Q_n\ \}\ \ s_2\ \ \{\ R_n\ ,\ R_e\ \}}$$

We have to prove:

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

By the first induction hypothesis we get:

$$\begin{array}{l}
\vdash \dfrac{\mathit{Tree}_1}{\mathcal{A} \vdash \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ R_e\ \}} \quad \wedge \\
(I_{l_a} ... I_{l_{a\_end}}) = \nabla_S \left(\dfrac{\mathit{tree}_1}{\mathcal{A} \vdash \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ R_e\ \}},\ l_a, l_b, l_{exc} \right) \ \wedge \\
\big(Q_n \quad \Rightarrow \quad E_{l_{a\_end+1}}\big) \quad \wedge \\
(\ (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \quad \Rightarrow \quad E_{l_{exc}}) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_{a\_end} : \vdash \{E_l\}\ I_l
\end{array}$$

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

$$Q_n \quad \Rightarrow \quad E_{l_{a\_end+1}} \tag{C.1}$$

$$(R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \quad \Rightarrow \quad E_{l_{exc}} \tag{C.2}$$

The second implication is proven by hypothesis. The precondition $E_{l_{a\_end+1}}$ 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:

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

The second induction hypothesis is:

$$\begin{array}{l}
\vdash \dfrac{\mathit{Tree}_2}{\mathcal{A} \vdash \{\ Q_n\ \}\ \ s_2\ \ \{\ R_n\ ,\ R_e\ \}} \quad \wedge \\
(I_{l_b} ... I_{l_{b\_end}}) = \nabla_S \left(\dfrac{\mathit{tree}_2}{\mathcal{A} \vdash \{\ Q_n\ \}\ \ s_2\ \ \{\ R_n\ ,\ R_e\ \}},\ l_b, l_{b+1}, l_{exc} \right) \ \wedge \\
\big(R_n \quad \Rightarrow \quad E_{l_{b+1}}\big) \quad \wedge \\
(\ (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \quad \Rightarrow \quad E_{l_{exc}}) \\
\Rightarrow \\
\forall\ l\ \in\ l_b\ ...\ l_{b\_end} : \vdash \{E_l\}\ I_l
\end{array}$$

To apply the second induction hypothesis we have to show:

$$Q_n \quad \Rightarrow \quad E_{l_{b+1}} \tag{C.3}$$

$$(R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \quad \Rightarrow \quad E_{l_{exc}} \tag{C.4}$$

These implications hold by hypothesis. Thus, we get:

$$\forall\ l\ \in\ l_b\ ...\ l_{b\_end} : \vdash \{E_l\}\ I_l$$

Finally, we join both results and we get:

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

□

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:

$$T_{S_1} \equiv \frac{Tree_1}{\mathcal{A} \vdash \Big\{\ P\ \wedge\ e\ \Big\}\ \ s_1\ \ \Big\{\ Q_n\ ,\ Q_e\ \Big\}}$$

$$T_{S_2} \equiv \frac{Tree_2}{\mathcal{A} \vdash \Big\{\ P\ \wedge\ \neg e\ \Big\}\ \ s_2\ \ \Big\{\ Q_n\ ,\ Q_e\ \Big\}}$$

We have to show:

$$\begin{array}{l}
\vdash \dfrac{T_{S_1} \qquad T_{S_2}}{\mathcal{A} \vdash \{\ P\ \}\ \begin{array}{l}\texttt{if}\ e\ \texttt{then}\ s_1 \\ \texttt{else}\ s_2\ \texttt{end}\end{array}\ \{\ Q_n\ ,\ Q_e\ \}} \quad \wedge \\
(I_{l_a}...I_{l_e}) = \nabla_S \left(\dfrac{T_{S_1} \qquad T_{S_2}}{\mathcal{A} \vdash \{\ P\ \}\ \begin{array}{l}\texttt{if}\ e\ \texttt{then}\ s_1 \\ \texttt{else}\ s_2\ \texttt{end}\end{array}\ \{\ Q_n\ ,\ Q_e\ \}}, l_a, l_{e+1}, l_{exc} \right) \wedge \\
\big(Q_n\ \Rightarrow\ E_{l_{e+1}}\big)\ \wedge \\
(\ (Q_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \Rightarrow\ E_{l_{exc}}) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_e : \vdash \{E_l\}\ I_l
\end{array}$$

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:

$$
\begin{array}{l}
\vdash \dfrac{\mathit{Tree}_1}{\mathcal{A} \vdash \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}} \quad \wedge \\
(I_{l_c} ... I_{l_{c\_end}}) = \nabla_S \left(\dfrac{\mathit{tree}_1}{\mathcal{A} \vdash \{\ P\ \}\ \ s_1\ \ \{\ Q_n\ ,\ Q_e\ \}},\ l_c, l_d, l_{exc} \right) \ \wedge \\
(Q_n \ \Rightarrow\ E_{l_d}) \ \ \wedge \\
(\ (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \ \Rightarrow\ \ E_{l_{exc}}) \\
\Rightarrow \\
\forall\ l\ \in\ l_c\ ...\ l_{c\_end} : \vdash \{E_l\}\ I_l
\end{array}
$$

Since E_{l_d} is equal to Q_n, and $(R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \Rightarrow\ E_{l_{exc}}$ holds by hypothesis, we can apply the induction hypothesis, and we prove:

$$\forall\ l\ \in\ l_c\ ...\ l_{c_{end}} : \vdash \{E_l\}\ I_l$$

The proof of the translation of s_2 is similar to the translation of s_1. Thus, we have proven:

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

□

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:

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

The translation of the expression e holds by Lemma 12. To prove that the instruction at l_b holds, we have to show:

$$\mathit{shift}(P)\ \wedge\ s(0) = e\ \ \mathit{implies}\ \ \mathit{wp}(\textsf{brtrue}\ l_{d+1})$$

Applying the definition of *wp*, we have to prove:

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

The first implication holds because $\mathit{shift}(E_{l_c}) = P \wedge \neg e$. The second implication holds by the hypothesis: $(P \ \wedge \ e\ ) \ \Rightarrow \ E_{l_{d+1}}$. 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:

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

□

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:

$$T_{S_1} \equiv \frac{\mathit{Tree}_1}{\mathcal{A} \vdash \{\ \neg e \ \wedge \ I \ \} \ \ s_1 \ \ \{\ I \ , \ R_e \ \}}$$

We have to prove:

$$\begin{array}{l}
\vdash \dfrac{T_{S_1}}{\mathcal{A} \vdash \left\{\ I\ \right\} \ \textbf{until} \ e \ \textbf{loop} \ s_1 \ \left\{\ I \ \wedge \ e \ , \ R_e\ \right\}} \quad \wedge \\
(I_{l_a} ... I_{l_d}) = \nabla_S \left(\dfrac{T_{S_1}}{\mathcal{A} \vdash \left\{\ I\ \right\} \ \textbf{until} \ e \ \textbf{loop} \ s_1 \ \left\{\ I \ \wedge \ e \ , \ R_e\ \right\}}, \ l_a, l_{d+1}, l_{exc} \right) \ \wedge \\
\big((I \ \wedge \ e) \ \Rightarrow \ E_{l_{d+1}}\big) \quad \wedge \\
(\ (R_e \ \wedge \ \mathit{excV} \neq \mathit{null} \ \wedge \ s(0) = \mathit{excV}) \ \ \Rightarrow \ \ E_{l_{exc}}) \\
\Rightarrow \\
\forall\ l \ \in \ l_a \ ... \ l_d : \vdash \{E_l\} \ I_l
\end{array}$$

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 $(I \ \wedge \ e) \ \Rightarrow \ E_{l_{d+1}}$

Now, we need to show that the translation of s_1 is valid. By the first induction hypothesis we get:

$$\begin{array}{l}
\vdash \dfrac{\mathit{Tree}_1}{\mathcal{A} \vdash \{\ \neg e\ \wedge\ I\ \}\ \ s_1\ \ \{\ I\ ,\ R_e\ \}} \quad \wedge \\
(I_{l_b} ... I_{l_{b\_end}}) = \nabla_S \left(\dfrac{\mathit{Tree}_1}{\mathcal{A} \vdash \{\ \neg e\ \wedge\ I\ \}\ \ s_1\ \ \{\ I\ ,\ R_e\ \}},\ l_b, l_c, l_{exc} \right) \wedge \\
(I\ \Rightarrow\ E_{l_c})\ \ \wedge \\
(\ (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \ \Rightarrow\ \ E_{l_{exc}}) \\
\Rightarrow \\
\forall\ l\ \in\ l_b\ ...\ l_{b\_end} : \vdash \{E_l\}\ I_l
\end{array}$$

The precondition at l_c (E_{l_c}) is I, so $I\ \Rightarrow\ E_{l_c}$. The implication

$$(R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \ \Rightarrow\ \ E_{l_{exc}}$$

holds by hypothesis. Then, we can apply the induction hypothesis and get:

$$\forall\ l\ \in\ l_b\ ...\ l_{b\_end} : \vdash \{E_l\}\ I_l$$

Thus, we prove

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

□

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:

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

We have to prove:

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

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:

$$\begin{array}{l}
\vdash \dfrac{Tree_1}{\mathcal{A} \vdash \left\{ \; I_r \; \right\} \; s_1 \; \left\{ \; Q_n \; , \; Q_e \; \right\}} \quad \wedge \\
(I_{l_a} ... I_{l_{a\_end}}) = \nabla_S \left(\dfrac{Tree_1}{\mathcal{A} \vdash \left\{ \; I_r \; \right\} \; s_1 \; \left\{ \; Q_n \; , \; Q_e \; \right\}}, \; l_a, l_b, l_c \right) \quad \wedge \\
(Q_n \;\; \Rightarrow \;\; E_{l_b}) \quad \wedge \\
(\; (Q_e \;\wedge\; excV \neq null \;\wedge\; s(0) = excV) \;\; \Rightarrow \;\; E_{l_c}) \\
\Rightarrow \\
\forall \; l \; \in \; l_a \; ... \; l_{a\_end} : \vdash \{E_l\} \; I_l
\end{array}$$

Due to the precondition E_{l_b} is Q_n and the precondition E_{l_c} is:

$$(Q_e \;\wedge\; excV \neq null \;\wedge\; s(0) = excV)$$

we can apply the first induction hypothesis and get:

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

The second induction hypothesis is:

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

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

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

The first implication holds because the precondition E_{l_e} is:

$$(Retry \Rightarrow I_r\ \wedge\ \neg Retry \Rightarrow R_e)$$

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

$$\{Q_n\}\ l_b : \mathsf{leave}\ l_{i+1}$$

holds by the definition of *wp* for 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 *wp*. For example, the instruction specification at label l_f is proven as follows:

$(Retry \Rightarrow I_r \;\wedge\; \neg Retry \Rightarrow R_e \;\wedge\; s(0) = Retry)$ *implies* wp (brfalse l_h, E_{l_g})
  [*definition of wp*]
$(Retry \Rightarrow I_r \;\wedge\; \neg Retry \Rightarrow R_e \;\wedge\; s(0) = Retry)$ *implies*
    $(\; s(0) \Rightarrow shift(E_{l_g}) \;\wedge\; \neg s(0) \Rightarrow shift(E_{l_h}))$
  [*definition of* E_{l_g} *and* E_{l_h}]
$(Retry \Rightarrow I_r \;\wedge\; \neg Retry \Rightarrow R_e \;\wedge\; s(0) = Retry)$ *implies*
    $(\; s(0) \Rightarrow shift(I_r) \;\wedge\; \neg s(0) \Rightarrow shift(R_e))$
  [*definition of shift, and* I_r *and* R_e *do not refer to the stack*]
$(Retry \Rightarrow I_r \;\wedge\; \neg Retry \Rightarrow R_e \;\wedge\; s(0) = Retry)$ *implies* $(\; s(0) \Rightarrow I_r \;\wedge\; \neg s(0) \Rightarrow R_e)$
  [*this holds since* $s(0) = Retry$]

Joining the proofs we get:

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

□

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:

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

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

$$Q'_n \;\equiv \left\{ \begin{array}{l} T@m\_done \;\wedge\; \neg T@m\_exc \wedge \\ \big(Q_n \vee (\; P'' \;\wedge\; Result = T\_M\_RES \;\wedge\; T@m\_result = T\_M\_RES \;)\big) \end{array} \right\}$$

$$Q'_e \;\equiv \{\; T@m\_done \;\wedge\; T@m\_exc \;\wedge\; (Q_e \;\vee P''') \;\}$$

Let T_{body} be the following proof tree:

$$T_{body} \equiv \frac{Tree_1}{\begin{array}{l}\mathcal{A}, \{P\} \;\; T@m \; \{Q'_n, \; Q'_e\} \vdash \\ \\ \left\{ \; P'[false/T@m\_done] \wedge \; T@m\_done \; \right\} \;\; body(T@m) \;\; \left\{ \; Q_n \; , \; Q_e \; \right\}\end{array}}$$

We have to prove:

$$\begin{array}{l}
\vdash \dfrac{T_{body}}{\mathcal{A} \vdash \left\{ \; P \; \right\} \;\; T@m \;\; \left\{ \; Q'_n \; , \; Q'_e \; \right\}} \quad \wedge \\
(I_{l_a} ... I_{l_r}) = \nabla_S \left(\dfrac{T_{body}}{\mathcal{A} \vdash \left\{ \; P \; \right\} \;\; T@m \;\; \left\{ \; Q'_n \; , \; Q'_e \; \right\}}, \; l_a, l_{r+1}, l_{exc} \right) \;\; \wedge \\
\left(Q'_n \;\; \Rightarrow \;\; E_{l_{r+1}}\right) \quad \wedge \\
(\; (Q'_e \;\; \wedge \;\; excV \neq null \;\; \wedge \;\; s(0) = excV) \;\; \Rightarrow \;\; E_{l_{exc}}) \\
\Rightarrow \\
\forall \; l \; \in \; l_a \; ... \; l_r : \vdash \{E_l\} \; I_l
\end{array}$$

The instruction specifications at labels $l_a, ..., l_d, \; l_f, ..., l_r$ hold by the definition of wp. For example, we prove the instruction specification at l_a as follows:

$$\begin{array}{l}
P \;\; implies \;\; wp \; (\textsf{ldsfld} \;\; T@m\_done, \;\; shift(P) \wedge s(0) = T@m\_done) \\
\quad [definition \;\; of \;\; wp] \\
P \;\; implies \;\; unshift(\; shift(P) \wedge s(0) = T@m\_done[T@m/s(0)] \;) \\
\quad [definition \;\; of \;\; replacement] \\
P \;\; implies \;\; unshift(\; shift(P) \wedge T@m = T@m\_done) \\
\quad [definition \;\; of \;\; unshift \;\; and \;\; shift] \\
P \;\; implies \;\; (P \wedge T@m = T@m\_done) \\
\square
\end{array}$$

The interesting case of the proof is the translation of T_{body}. By the induction hypothesis we get:

$$
\begin{array}{l}
\vdash T_{body} \quad \wedge \\
(I_{l_e} ... I_{l_{e\_end}}) = \nabla_S\,(T_{body},\ l_e, l_f, l_i\ )\ \wedge \\
\left(Q_n \ \Rightarrow \ E_{l_f}\right) \quad \wedge \\
(\ (\mathit{shift}(Q_e)\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow \ E_{l_i}) \\
\Rightarrow \\
\forall\ l\ \in\ l_e\ ...\ l_{e\_end} : \vdash \{E_l\}\ I_l
\end{array}
$$

To be able to apply the induction hypothesis, we need to show:

$$
\begin{array}{r}
\left(Q_n \ \Rightarrow \ E_{l_f}\right) \\
(\ (\mathit{shift}(Q_e)\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow \ E_{l_i})
\end{array}
\tag{C.5}
$$

The precondition at label E_{l_f} is Q_n, so the first implication holds. The precondition E_{l_i} is:

$$(\mathit{shift}(Q_e)\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)$$

then the second implication also holds, and we can apply the induction hypothesis and get:

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

Finally, joining the proofs we get:

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

□

Appendix D

Soundness Proof of the Java 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)

$$\vdash \frac{Tree_1}{\left\{ \; P \; \right\} \quad s_1 \quad \left\{ \; Q_n, Q_b, Q_e \; \right\}} \quad \wedge$$

$$(I_{l_{start}} ... I_{l_{end}}) = \nabla_S \left(\frac{Tree_1}{\left\{ \; P \; \right\} \quad s_1 \quad \left\{ \; Q_n, Q_b, Q_e \; \right\}}, \; l_{start}, l_{end+1}, l_{break}, m \right) \quad \wedge$$

$$\left(Q_n \quad \Rightarrow \quad E_{l_{end+1}}\right) \quad \wedge$$

$$(Q_b \quad \Rightarrow \quad E_{l_{break}}) \quad \wedge$$

$$(\; \forall T : \mathit{Type} : T \preceq \mathit{Throwable} : (Q_e \;\wedge\; excV \neq null \;\wedge\; s(0) = excV) \quad \Rightarrow \quad E_{m[T]})$$

$$\Rightarrow$$

$$\forall \; l \; \in \; l_{start} \; ... \; l_{end} : \vdash \{E_l\} \; I_l$$

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:

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

Replacing the compound rule in Theorem 6, we have to prove:

$$\vdash \frac{T_{S_1} \qquad T_{S_2}}{\left\{ \; P \; \right\} \quad s_1; s_2 \quad \left\{ \; R_n, R_b, R_e \; \right\}} \quad \wedge$$

$$(I_{l_a} ... I_{l_b}) = \nabla_S \left(\frac{T_{S_1} \qquad T_{S_2}}{\left\{ \; P \; \right\} \quad s_1; s_2 \quad \left\{ \; R_n, R_b, R_e \; \right\}}, \; l_a, l_{b+1}, l_{break}, l_{exc}, \; m \right) \quad \wedge$$

$$\left(R_n \quad \Rightarrow \quad E_{l_{b+1}}\right) \quad \wedge$$

$$(R_b \quad \Rightarrow \quad E_{l_{break}}) \quad \wedge$$

$$(\; \forall T : \mathit{Type} : T \preceq \mathit{Throwable} : (R_e \;\wedge\; excV \neq null \;\wedge\; s(0) = excV) \quad \Rightarrow \quad E_{m[T]})$$

$$\Rightarrow$$

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

By the first induction hypothesis we get:

$$
\begin{array}{l}
\vdash \dfrac{Tree_1}{\left\{\ P\ \right\}\ s_1\ \left\{\ Q_n, R_b, R_e\ \right\}} \quad \wedge \\
(I_{l_a} ... I_{l_{a\_end}}) = \nabla_S \left(\dfrac{tree_1}{\left\{\ P\ \right\}\ s_1\ \left\{\ Q_n, R_b, R_e\ \right\}},\ l_a, l_b, l_{break}, l_{exc},\ m \right) \wedge \\
(Q_n \ \Rightarrow\ E_{l_b}) \quad \wedge \\
(R_b \ \Rightarrow\ E_{l_{break}}) \quad \wedge \\
(\ \forall T : Type : T \preceq Throwable : (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{m[T]}) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_{a\_end} : \vdash \{E_l\}\ I_l
\end{array}
$$

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

$$Q_n \ \Rightarrow\ E_{l_b} \quad \text{(D.1)}$$
$$R_b \ \Rightarrow\ E_{l_{break}} \quad \text{(D.2)}$$
$$\forall T : Type : T \preceq Throwable : (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{m[T]} \quad \text{(D.3)}$$

The second and the third implications are proven by hypothesis since the break label l_{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:

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

The second induction hypothesis is:

$$
\begin{array}{l}
\vdash \dfrac{Tree_2}{\left\{\ Q_n\ \right\}\ s_2\ \left\{\ R_n, R_b, R_e\ \right\}} \quad \wedge \\
(I_{l_b} ... I_{l_{b\_end}}) = \nabla_S \left(\dfrac{tree_2}{\left\{\ Q_n\ \right\}\ s_2\ \left\{\ R_n, R_b, R_e\ \right\}},\ l_b, l_{b+1}, l_{break}, l_{exc},\ m \right) \wedge \\
\left(R_n \ \Rightarrow\ E_{l_{b+1}}\right) \quad \wedge \\
(R_b \ \Rightarrow\ E_{l_{break}}) \quad \wedge \\
(\ \forall T : Type : T \preceq Throwable : (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{m[T]}) \\
\Rightarrow \\
\forall\ l\ \in\ l_b\ ...\ l_{b\_end} : \vdash \{E_l\}\ I_l
\end{array}
$$

To apply this hypothesis, we need to show:

$$R_n \quad \Rightarrow \quad E_{l_{b+1}} \tag{D.4}$$
$$R_b \quad \Rightarrow \quad E_{l_{break}} \tag{D.5}$$
$$\forall T : \mathit{Type} : T \preceq \mathit{Throwable} : (R_e \ \wedge \ \mathit{excV} \neq \mathit{null} \ \wedge \ s(0) = \mathit{excV}) \quad \Rightarrow \quad E_{m[T]} \tag{D.6}$$

These implications holds by hypothesis. So, we get:

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

Finally, we join both results and we prove:

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

□

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_{while} be the following proof trees:

$$T_{S_1} \equiv \frac{\mathit{Tree}_1}{\{ \ e \ \wedge \ I \ \} \quad s_1 \quad \{ \ I, Q_b, R_e \ \}}$$

$$T_{while} \equiv \frac{T_{S_1}}{\{ \ I \ \} \ \ \texttt{while} \ (e) \ s_1 \ \ \{ \ (I \wedge \neg e) \vee \ Q_b, \mathit{false}, R_e \ \}}$$

We have to show:

$\vdash T_{while} \quad \wedge$

$(I_{l_a} ... I_{l_d}) = \nabla_S \, (T_{while}, \ l_a, l_{d+1}, l_{break}, l_{exc}, \ m \ ) \ \wedge$

$\big((I \wedge \neg e) \vee \ Q_b \quad \Rightarrow \quad E_{l_{d+1}}\big) \quad \wedge$

$(\mathit{false} \quad \Rightarrow \quad E_{l_{break}}) \quad \wedge$

$(\ \forall T : \mathit{Type} : T \preceq \mathit{Throwable} : (R_e \ \wedge \ \mathit{excV} \neq \mathit{null} \ \wedge \ s(0) = \mathit{excV}) \quad \Rightarrow \quad E_{m[T]})$

$\Rightarrow$

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

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:

$$\begin{array}{l}
\vdash T_{S_1} \quad \wedge \\
(I_{l_b} ... I_{l_{b\_end}}) = \nabla_S\,(T_{S_1},\ l_b, l_c, l_{d+1}, l_{exc},\ m\ ) \ \wedge \\
(I \quad \Rightarrow \quad E_{l_c}) \quad \wedge \\
\left(Q_b \quad \Rightarrow \quad E_{l_{d+1}}\right) \quad \wedge \\
(\ \forall T : \mathit{Type} : T \preceq \mathit{Throwable} : (R_e \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \quad \Rightarrow \quad E_{m[T]}) \\
\Rightarrow \\
\forall\ l \ \in \ l_b \ ... \ l_{b\_end} : \vdash \{E_l\}\ I_l
\end{array}$$

To apply this hypothesis, we need to show:

$$I \quad \Rightarrow \quad E_{l_c} \tag{D.7}$$

$$Q_b \quad \Rightarrow \quad E_{l_{d+1}} \tag{D.8}$$

$$\forall T : \mathit{Type} : T \preceq \mathit{Throwable} : (R_e \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \quad \Rightarrow \quad E_{m[T]} \tag{D.9}$$

The first implication holds since E_{l_c} is equal to I. The second implication holds from the hypothesis

$$(I \wedge \neg e) \vee \ Q_b \quad \Rightarrow \quad E_{l_{d+1}}$$

And the third implication is true by hypothesis. Then, applying the induction hypothesis we get:

$$\forall\ l \ \in \ l_b \ ... \ l_{b\_end} : \vdash \{E_l\}\ I_l$$

Joining the proofs, we show:

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

□

D.1.4 Break Rule

The break rule has been translated in Section 8.1.4 on page 158. The goal to prove is:

$$\vdash \frac{}{\left\{ P \right\} \ \texttt{break} \ \left\{ \mathit{false}, P, \mathit{false} \right\}} \quad \wedge$$
$$(I_{l_{start}} = \nabla_S \left(\frac{}{\left\{ P \right\} \ \texttt{break} \ \left\{ \mathit{false}, P, \mathit{false} \right\}}, \ l_{start}, l_{start+1}, l_{break}, l_{exc}, \ m \right) \ \wedge$$
$$\big(\mathit{false} \ \Rightarrow \ E_{l_{start+1}}\big) \quad \wedge$$
$$(P \ \Rightarrow \ E_{l_{break}}) \quad \wedge$$
$$(\ \forall T : \mathit{Type} : T \preceq \mathit{Throwable} : (\mathit{false} \ \wedge \ \mathit{excV} \neq \mathit{null} \ \wedge \ s(0) = \mathit{excV}) \ \Rightarrow \ E_{m[T]})$$
$$\Rightarrow$$
$$\{P\} \quad l_{start} : \mathsf{br} \ l_{break}$$

By definition of wp, we have to show:

$$P \Rightarrow E_{l_{break}}$$

The implication holds by hypothesis.
□

D.1.5 Throw Rule

The translation of this rule is presented in Section 8.1.5 on page 158.
We have to prove:

$$\vdash \frac{}{\left\{ P[e/\mathit{excV}] \right\} \ \texttt{throw} \ e \ \left\{ \mathit{false}, \mathit{false}, P \right\}} \quad \wedge$$
$$(I_{l_a} ... I_{l_b}) = \nabla_S \left(\frac{}{\left\{ P[e/\mathit{excV}] \right\} \ \texttt{throw} \ e \ \left\{ \mathit{false}, \mathit{false}, P \right\}}, \ l_a, l_{b+1}, l_{break}, l_{exc}, \ m \right) \ \wedge$$
$$\big(\mathit{false} \ \Rightarrow \ E_{l_{b+1}}\big) \quad \wedge$$
$$(\mathit{false} \ \Rightarrow \ E_{l_{break}}) \quad \wedge$$
$$(\ \forall T : \mathit{Type} : T \preceq \mathit{Throwable} : (P \ \wedge \ \mathit{excV} \neq \mathit{null} \ \wedge \ s(0) = \mathit{excV}) \ \Rightarrow \ E_{m[T]})$$
$$\Rightarrow$$
$$\forall \ l \ \in \ l_a \ ... \ l_b : \vdash \{E_l\} \ I_l$$

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:

$$shift(P[e/excV]) \ \wedge \ s(0) = e \quad implies \quad wp(\mathsf{throw}, \ E_{m[T]})$$

This implication holds by the hypothesis:

$$\forall T : Type : T \preceq Throwable : (P \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \quad \Rightarrow \quad E_{m[T]}$$

□

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:

$$T_{S_1} \equiv \frac{Tree_1}{\left\{ \ P \ \right\} \ \ s_1 \ \ \left\{ \ Q_n, Q_b, Q \ \right\}}$$

$$T_{S_2} \equiv \frac{Tree_2}{\left\{ \ Q'_e[e/excV] \ \right\} \ \ s_2 \ \ \left\{ \ Q_n, Q_b, R_e \ \right\}}$$

where

$$Q \equiv (\ (Q''_e \ \wedge \ \tau(excV) \npreceq T) \vee (Q'_e \ \wedge \ \tau(excV) \preceq T) \ )$$

Let R be the following postcondition:

$$R \equiv (R_e \ \vee \ (Q''_e \ \wedge \ \tau(excV) \npreceq T) \ )$$

We need to show:

$$\vdash \dfrac{T_{S_1} \qquad T_{S_2}}{\Big\{\ P\ \Big\}\ \ \texttt{try}\ s_1\ \texttt{catch}\ (T\ \ e)\ s_2\ \ \Big\{\ Q_n, Q_b, R\ \Big\}} \quad \wedge$$
$$(I_{l_a} ... I_{l_e}) = \nabla_S \left(\dfrac{T_{S_1} \qquad T_{S_2}}{\Big\{\ P\ \Big\}\ \ \texttt{try}\ s_1\ \texttt{catch}\ (T\ \ e)\ s_2\ \ \Big\{\ Q_n, Q_b, R\ \Big\}},\ l_a, l_{e+1}, l_{break}, l_{exc},\ m \right) \ \wedge$$
$$\big(Q_n \quad \Rightarrow \quad E_{l_{e+1}}\big) \quad \wedge$$
$$(Q_b \quad \Rightarrow \quad E_{l_{break}}) \quad \wedge$$
$$(\ \forall T : \mathit{Type} : T \preceq \mathit{Throwable} : (R\ \wedge\ \mathit{excV} \neq \mathit{null}\ \wedge\ s(0) = \mathit{excV}) \quad \Rightarrow \quad E_{m[T]})$$
$$\Rightarrow$$
$$\forall\ l\ \in\ l_a\ ...\ l_e : \vdash \{E_l\}\ I_l$$

Applying the first induction hypothesis we get:

$$\vdash \dfrac{\mathit{Tree}_1}{\{\ P\ \}\ \ s_1\ \ \{\ Q_n, Q_b, Q\ \}} \quad \wedge$$
$$(I_{l_a} ... I_{l_{a\_end}}) = \nabla_S \left(\dfrac{\mathit{Tree}_1}{\{\ P\ \}\ \ s_1\ \ \{\ Q_n, Q_b, Q\ \}},\ l_a, l_b, l_{break}, l_{exc}, m' \right) \ \wedge$$
$$(Q_n \quad \Rightarrow \quad E_{l_b}) \quad \wedge$$
$$(Q_b \quad \Rightarrow \quad E_{l_{break}}) \quad \wedge$$
$$(\ \forall T : \mathit{Type} : T \preceq \mathit{Throwable} : (Q\ \wedge\ \mathit{excV} \neq \mathit{null}\ \wedge\ s(0) = \mathit{excV}) \quad \Rightarrow \quad E_{m'[T]})$$
$$\Rightarrow$$
$$\forall\ l\ \in\ l_a\ ...\ l_{a\_end} : \vdash \{E_l\}\ I_l$$

where m' is defined as $m\ \big[\ \mathsf{T}\ \ \rightarrow l_c\ \big]$
To apply the induction hypothesis, we have to show:

$$Q_n \quad \Rightarrow \quad E_{l_b} \quad \text{(D.10)}$$
$$Q_b \quad \Rightarrow \quad E_{l_{break}} \quad \text{(D.11)}$$
$$\forall T : \mathit{Type} : T \preceq \mathit{Throwable} : (Q\ \wedge\ \mathit{excV} \neq \mathit{null}\ \wedge\ s(0) = \mathit{excV}) \quad \Rightarrow \quad E_{m'[T]} \quad \text{(D.12)}$$

The first two implications holds by hypothesis and the definition of E_{l_b}. In the third implication, m' is defined as $m\ \big[\ \mathsf{T}\ \ \rightarrow l_c\ \big]$. Then the implication holds for all exception types except for T from the hypothesis:

$$\forall T : \mathit{Type} : T \preceq \mathit{Throwable} : (R\ \wedge\ \mathit{excV} \neq \mathit{null}\ \wedge\ s(0) = \mathit{excV}) \quad \Rightarrow \quad E_{m[T]}$$

In the case of T, $m'[T]$ is E_{l_c}. By the definition of E_{l_c}, we need to prove:

$$(Q \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \ \Rightarrow \begin{pmatrix} shift(Q'_e) \ \wedge \ excV \neq null \\ \wedge \ \tau(excV) \preceq T \ \wedge \ s(0) = excV \end{pmatrix}$$

Since the exception type is T, then $\tau(excV) \preceq T$. Then by definition of Q we have:

$$\begin{pmatrix} Q'_e \ \wedge \ \tau(excV) \preceq T \ \wedge \\ excV \neq null \ \wedge \ s(0) = excV \end{pmatrix} \Rightarrow \begin{pmatrix} shift(Q'_e) \ \wedge \ excV \neq null \\ \wedge \ \tau(excV) \preceq T \ \wedge \ s(0) = excV \end{pmatrix}$$

By definition of *shift*, $shift(Q'_e) = Q'_e$ because Q'_e does not refer to the stack. Then, the implication holds, and we get by the first induction hypothesis:

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

The instructions at l_b, l_c, and l_e are proven by hypothesis and the definition of wp. The second induction hypothesis is:

$$\begin{array}{l}
\vdash \dfrac{Tree_2}{\{ \ Q'_e[e/excV] \ \} \quad s_2 \quad \{ \ Q_n, Q_b, R_e \ \}} \quad \wedge \\
(I_{l_d} ... I_{l_{d\_end}}) = \nabla_S \left(\dfrac{Tree_1}{\{ \ Q'_e[e/excV] \ \} \quad s_2 \quad \{ \ Q_n, Q_b, R_e \ \}}, \ l_d, l_e, l_{break}, l_{exc}, m \right) \ \wedge \\
(Q_n \quad \Rightarrow \quad E_{l_e}) \quad \wedge \\
(Q_b \quad \Rightarrow \quad E_{l_{break}}) \quad \wedge \\
(\ \forall T : \ Type : \ T \preceq \ Throwable : (R_e \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \quad \Rightarrow \quad E_{m[T]}) \\
\Rightarrow \\
\forall \ l \ \in \ l_d \ ... \ l_{d\_end} : \vdash \{E_l\} \ I_l
\end{array}$$

To apply this induction hypothesis, we need to show that the following implications hold:

$$Q_n \quad \Rightarrow \quad E_{l_e} \tag{D.13}$$

$$Q_b \quad \Rightarrow \quad E_{l_{break}} \tag{D.14}$$

$$\forall T : \ Type : \ T \preceq \ Throwable : (R_e \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \quad \Rightarrow \quad E_{m[T]} \tag{D.15}$$

Since the precondition E_{l_e} is Q_n, the first implication holds. The second and the third implication hold by hypothesis. Then we get

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

Joining the proofs we show:

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

□

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)

$$
\left(\begin{array}{l}
\vdash \dfrac{Tree}{\left\{\ P\ \right\}\ \ s_1\ \ \left\{\ Q_n, Q_b, Q_e\ \right\}} \triangleq\ T_{S_1}\quad \wedge \\
[(I_{l_{start}} ... I_{l_{end}}), et] = \nabla_S\,(T_{S_1},\ l_{start}, l_{end+1}, l_{break}, f, et') \wedge \\
(\forall\ T : \mathit{Type} : (T \preceq \mathit{Throwable} \vee T \equiv \mathit{any}) : \\
\ (Q_e \wedge \mathit{excV} \neq \mathit{null} \wedge\ s(0) = \mathit{excV})\ \Rightarrow\ E_{et'[l_{start}, l_{end}, T]}) \wedge \\
(Q_n\ \ \Rightarrow\ \ E_{l_{end+1}})\ \ \wedge \\
(Q_b\ \Rightarrow (f = \emptyset\ \Rightarrow\ (Q_b\ \Rightarrow\ E_{l_{break}}))\ \wedge \\
\qquad \left(\begin{array}{l}
f \neq \emptyset \Rightarrow \forall i\ \in\ 1..k : \\
\left(\begin{array}{l}
(\vdash \{U^i\}\ s_i\ \{V^i\}\quad \wedge\ \ (V_b^i \Rightarrow U_b^{i+1}) \wedge \\
(\ \forall T : \mathit{Type} : ((T \preceq \mathit{Throwable}) \vee (T \equiv \mathit{any})) : \\
\qquad (V_e^i\ \wedge\ \mathit{excV} \neq \mathit{null}\ \wedge\ s(0) = \mathit{excV})\ \ \Rightarrow\ \ E_{et_i[T]}\ ) \wedge \\
(Q_b \Rightarrow U_b^1)\ \wedge\ (V_b^k \Rightarrow E_{l_{break}}) \wedge \\
(\forall et_i = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset) \wedge (et_i \subseteq et'))
\end{array}\right)
\end{array}\right)
\end{array}\right)
$$

$$
\begin{array}{l}
\Rightarrow \\
\forall\ l\ \in\ l_{start}\ ...\ l_{end} : \vdash \{E_l\}\ I_l
\end{array}
$$

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

$$\nabla_S\left(\frac{Tree}{\{\ P\ \}\quad s\quad\{\ Q_n, Q_b, Q_e\ \}},\ l_a,\ l_{b+1},\ l_{break},\ f,\ et\right)\ =\ [(I_{l_a}...I_{l_b}), et']$$

and $l_{start} \le l_a < l_b \le l_{end}$ *then,*
for every l_s, l_e *: Label such that* $l_b < l_s < l_e \le l_{end}$*, and*
for every T *: Type such that* $T \preceq Throwable \vee T \equiv any$*, the following holds:*

$$et[l_{start}, l_{end}, T] = et'[l_s, l_e, T]$$

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 ∇_S, it only can have the structure: $et' = et + et_a$ where et_a is a new exception table created by ∇_S or $et' = divide(et, r, l'_a, l'_b)$ 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 $l_a \le l'_a < l'_b \le l_b$. This property holds because ∇_S only adds new lines to the exception table taken as parameter (et) in the translation of `try-catch` and `try-finally`, and ∇_S only divides the table in the translation of `break`.

Case $\mathbf{et' = et + et_a}$
The exception table et_a is produced by:

$$\nabla_S\left(\frac{tree}{\{\ P\ \}\quad s\quad\{\ Q_n, Q_b, Q_e\ \}},\ l_a,\ l_{b+1},\ l_{break},\ f, et\right)$$

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 $l_b < l_s < l_e \le l_{end}$. 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:

$$\forall\ l_s, l_e : l_b < l_s < l_e \le l_{end}$$

then $l_{start} \le l_s < l_e \le l_{end}$ 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_{start} and l_{end}

$(et[l_{start}, l_{end}, T])$. Then $et[l_s, l_e, T] = et[l_{start}, l_{end}, T]$. So, this proves that $et'[l_s, l_e, T] = et[l_{start}, l_{end}, T]$.

Case $\mathbf{et' = divide(et, r, l'_a, l'_b)}$

The definition of the function *divide* has been presented in Section 8.2.5 on page 167. There are two possible results after the execution of $divide(et, r, l'_a, l'_b)$ either *divide* does not modify the exception entry $[l_{start'}, l_{end'}, l_t, T]$ in $et[l_{start}, l_{end}, T]$ with $l_{start'} \leq l_{start}$ and $l_{end'} \leq l_{end}$) or it does.

If $divide(et, r, l'_a, l'_b)$ does not modify the exception entry $[l_{start'}, l_{end'}, l_t, T]$, then using a similar reasoning to the above case, we can conclude that $et'[l_s, l_e, T] = et[l_{start}, l_{end}, T]$.

If $divide(et, r, l'_a, l'_b)$ modifies the exception entry $[l_{start'}, l_{end'}, l_t, T]$ then it will be divided as:

$$[l_{start'}, l_{a'}, l_t, T] \ + \ [l_{b'}, l_{end'}, l_t, T]$$

We are looking for the target label between l_s and l_e. But we know $l_b < l_s < l_e \leq l_{end}$ and $l_a \leq l'_a < l'_b \leq l_b$. Then $et'[l_s, l_e, T]$ will return the target label in the divided line $[l_{b'}, l_{end'}, l_t, T]$, which target label is the same than $et[l_{start}, l_{end}, T]$. So, we can conclude $et[l_{start}, l_{end}, T] = et'[l_s, l_e, T]$.

□

Lemma 4 (Section 8.2.5) *Let r : ExcTableEntry and et' : ExcTable be such that $r \in et'$.*
If et : ExcTable and l_s, l_e : Label are such that $et = divide(et', r, l_s, l_e)$, then

$$et[l_s, l_e, T] = r[T]$$

Proof of Lemma 4 (Section 8.2.5)

The proof runs by induction on et'.

Base case: $et' = [\,]$

By definition of *divide*, $divide([\,], r, l_s, l_e) = [r]$. Then the lemma holds.

Inductive Case: $divide(r_i + et'_1, r, l_s, l_e)$

If $r \not\subseteq r_i$ then the function *divide* either updates r_i or not depending on whether $(r.from > r_i.to \ \wedge \ r.to < r_i.to)$ holds or not. But, it does not update r. Then by induction hypothesis we get $divide(r_i + et'_1, r, l_s, l_e)[l_s, l_e, T] = r[T]$.

If $r \subseteq r_i$ then the function *divide* returns $r_i + et'_1$. Then $r_i + et'_1[l_s, l_e, T] = r[T]$ because $r \subseteq r_i$ and *divide* does not change r_i.

□

Following, we present the soundness proof for compound, while, try-finally, and break rules.

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:

$$T_{S_1} \equiv \frac{\mathit{Tree}_1}{\{\ P\ \}\quad s_1 \quad \{\ Q_n, R_b, R_e\ \}} \qquad T_{S_2} \equiv \frac{\mathit{Tree}_2}{\{\ Q_n\ \}\quad s_2 \quad \{\ R_n, R_b, R_e\ \}}$$

Replacing the compound rule in Theorem 7, we have to prove:

$$\begin{array}{l}
\vdash \dfrac{T_{S_1} \qquad\qquad T_{S_2}}{\left\{\ P\ \right\}\quad s_1; s_2 \quad \left\{\ R_n, R_b, R_e\ \right\}} \quad \wedge \\
[(I_{l_a}...I_{l_b}),\ et_2] = \nabla_S \left(\dfrac{T_{S_1} \qquad\qquad T_{S_2}}{\left\{\ P\ \right\}\quad s_1; s_2 \quad \left\{\ R_n, R_b, R_e\ \right\}},\ l_a,\ l_{b+1},\ l_{break},\ f, et \right) \quad \wedge \\
(\forall\ T : \mathit{Type} : ((T \preceq \mathit{Throwable}) \vee (T \equiv \mathit{any})) : \\
\qquad\qquad (R_e\ \wedge\ \mathit{excV} \neq \mathit{null}\ \wedge\ s(0) = \mathit{excV}) \ \Rightarrow\ E_{et[l_a, l_b, T]}) \quad \wedge \\
\left(R_n \quad \Rightarrow \quad E_{l_{b+1}}\right)\ \wedge \\
(R_b\ \Rightarrow (f = \emptyset\ \Rightarrow\ (R_b\ \Rightarrow\ E_{l_{break}}))\ \wedge \\
\qquad \left(\begin{array}{l}
f \neq \emptyset \Rightarrow \forall i\ \in\ 1..k : \\
\left(\begin{array}{l}
(\vdash \{U^i\}\ s_i\ \{V^i\} \quad \wedge \quad (V_b^i \Rightarrow U_b^{i+1})\ \wedge \\
(\ \forall T : \mathit{Type} : ((T \preceq \mathit{Throwable}) \vee (T \equiv \mathit{any})) : \\
\qquad (\ V_e^i\ \wedge\ \mathit{excV} \neq \mathit{null}\ \wedge\ s(0) = \mathit{excV}) \quad \Rightarrow \quad E_{et_i[T]}\ )\ \wedge \\
(R_b \Rightarrow U_b^1)\ \wedge\ (V_b^k \Rightarrow E_{l_{break}})\ \wedge \\
(\forall et_i = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset) \wedge (et_i \subseteq et))
\end{array} \right)
\end{array} \right) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_b : \vdash \{E_l\}\ I_l
\end{array}$$

By the first induction hypothesis, we get:

$$
\begin{array}{l}
\vdash T_{S_1} \quad \wedge \\
[(I_{l_a} ... I_{l_{a\_end}}),\ et_1] = \nabla_S\, (T_{S_1},\ l_a,\ l_b,\ l_{break},\ f, et\ ) \ \wedge \\
(\forall\ T : \mathit{Type} : ((T \preceq \mathit{Throwable}) \vee (T \equiv \mathit{any})) : \\
\qquad\qquad (R_e\ \wedge\ \mathit{excV} \neq \mathit{null}\ \wedge\ s(0) = \mathit{excV}) \quad \Rightarrow \quad E_{et[l_a, l_{a\_end}, T]}) \quad \wedge \\
(Q_n \quad \Rightarrow \quad E_{l_b}) \ \wedge \\
(R_b\ \Rightarrow (f = \emptyset\ \Rightarrow\ (R_b\ \Rightarrow\ E_{l_{break}})) \ \wedge \\
\qquad \left(\begin{array}{l}
f \neq \emptyset \Rightarrow \forall i\ \in\ 1..k : \\
\left(\begin{array}{l}
(\vdash \{U^i\}\ s_i\ \{V^i\} \quad \wedge\ \ (V_b^i \Rightarrow U_b^{i+1}) \ \wedge \\
(\ \forall T : \mathit{Type} : ((T \preceq \mathit{Throwable}) \vee (T \equiv \mathit{any})) : \\
\qquad (V_e^i\ \wedge\ \mathit{excV} \neq \mathit{null}\ \wedge\ s(0) = \mathit{excV}) \quad \Rightarrow \quad E_{et_i[T]}\ ) \ \wedge \\
(R_b \Rightarrow U_b^1)\ \wedge\ (V_b^k \Rightarrow E_{l_{break}}) \ \wedge \\
(\forall et_i = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset) \wedge (et_i \subseteq et))
\end{array} \right)
\end{array} \right) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_{a\_end} : \vdash \{E_l\}\ I_l
\end{array}
$$

To be able to apply the first induction hypothesis we have to show that $Q_n \ \Rightarrow\ E_{l_b}$. The implication is true because the precondition E_{l_b} is Q_n. Furthermore, we have to prove:

$$
\begin{array}{l}
(\forall\ T : \mathit{Type} : ((T \preceq \mathit{Throwable}) \vee (T \equiv \mathit{any})) : \\
\qquad\qquad (R_e\ \wedge\ \mathit{excV} \neq \mathit{null}\ \wedge\ s(0) = \mathit{excV}) \quad \Rightarrow \quad E_{et[l_a, l_{a\_end}, T]})
\end{array}
$$

We can prove this implication using the theorem hypothesis and the fact that $l_{a\_end} < l_b$. Finally, we need to show that the *finally properties* holds, and this property is proven by hypothesis. Then, we get:

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

The second induction hypothesis is:

$$
\begin{array}{l}
\vdash T_{S_2} \quad \wedge \\
[(I_{l_b} ... I_{l_{b\_end}}),\ et_2] = \nabla_S\,(T_{S_2},\ l_b,\ l_{b+1},\ l_{break},\ f, et_1\ ) \ \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \quad \Rightarrow \quad E_{et_1[l_b, l_{b\_end}, T]}) \quad \wedge \\
\big(R_n \quad \Rightarrow \quad E_{l_{b+1}}\big) \ \wedge \\
(R_b\ \Rightarrow (f = \emptyset\ \Rightarrow\ (R_b\ \Rightarrow\ E_{l_{break}})) \ \wedge \\
\qquad \left(\begin{array}{l}
f \neq \emptyset \Rightarrow \forall i\ \in\ 1..k : \\
\left(\begin{array}{l}
(\vdash \{U^i\}\ s_i\ \{V^i\} \quad \wedge\ \ (V_b^i \Rightarrow U_b^{i+1}) \ \wedge \\
(\ \forall T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (V_e^i\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \quad \Rightarrow \quad E_{et_i[T]}\ ) \ \wedge \\
(R_b \Rightarrow U_b^1)\ \wedge\ (V_b^k \Rightarrow E_{l_{break}}) \ \wedge \\
(\forall et_i = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset) \wedge (et_i \subseteq et_1))
\end{array}\right)
\end{array}\right) \\
\Rightarrow \\
\forall\ l\ \in\ l_b\ ...\ l_{b\_end} : \vdash \{E_l\}\ I_l
\end{array}
$$

The implication $R_n \quad \Rightarrow \quad E_{l_{b+1}}$ holds by hypothesis. Furthermore, the implication

$$
\begin{array}{l}
\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \quad \Rightarrow \quad E_{et_1[l_b, l_{b\_end}, T]}
\end{array}
$$

holds by the hypothesis:

$$
\begin{array}{l}
\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \quad \Rightarrow \quad E_{et[l_a, l_b, T]}
\end{array}
$$

and Lemma 3. The *finally properties* holds the hypothesis:

$$
\begin{array}{l}
(R_b\ \Rightarrow (f = \emptyset\ \Rightarrow\ (R_b\ \Rightarrow\ E_{l_{break}})) \ \wedge \\
\qquad \left(\begin{array}{l}
f \neq \emptyset \Rightarrow \forall i\ \in\ 1..k : \\
\left(\begin{array}{l}
(\vdash \{U^i\}\ s_i\ \{V^i\} \quad \wedge\ \ (V_b^i \Rightarrow U_b^{i+1}) \ \wedge \\
(\ \forall T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (V_e^i\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \quad \Rightarrow \quad E_{et_i[T]}\ ) \ \wedge \\
(R_b \Rightarrow U_b^1)\ \wedge\ (V_b^k \Rightarrow E_{l_{break}}) \ \wedge \\
(\forall et_i = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset) \wedge (et_i \subseteq et))
\end{array}\right)
\end{array}\right)
\end{array}
$$

and Lemma 3. Thus, we can apply the second induction hypothesis and prove:

$$\forall\ l\ \in\ l_b\ ...\ l_{b\_end} : \vdash \{E_l\}\ I_l$$

Finally, we join both result and we get:

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

□

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:

$$T_{S_1} \equiv \frac{Tree_1}{\{\ e\ \wedge\ I\ \}\quad s_1\quad \{\ I, Q_b, R_e\ \}}$$

We have to prove:

$$\vdash \frac{T_{S_1}}{\left\{\ I\ \right\}\ \texttt{while}\ (e)\ s_1\ \left\{\ (I \wedge \neg e) \vee Q_b, false, R_e\ \right\}} \quad \wedge$$

$$[(I_{l_a}...I_{l_d}), et_1] =$$

$$\nabla_S \left(\frac{T_{S_1}}{\left\{\ I\ \right\}\ \texttt{while}\ (e)\ s_1\ \left\{\ (I \wedge \neg e) \vee Q_b, false, R_e\ \right\}},\ l_a,\ l_{d+1},\ l_{break},\ f,\ et \right) \wedge$$

$$(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) :$$
$$(R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{et[l_a,l_d,T]}) \ \wedge$$
$$\big(((I_n\ \wedge \neg e)\ \vee\ Q_b\ )\ \Rightarrow\ E_{l_{d+1}}\big) \wedge$$
$$(false\ \Rightarrow (f = \emptyset\ \Rightarrow\ (false\ \Rightarrow\ E_{l_{break}})) \wedge$$

$$\left(\begin{array}{l} f \neq \emptyset \Rightarrow \forall i\ \in\ 1..k : \\ \left(\begin{array}{l} (\vdash \{U^i\}\ s_i\ \{V^i\} \quad \wedge\ \ (V_b^i \Rightarrow U_b^{i+1}) \wedge \\ (\ \forall T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\ \qquad (V_e^i\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{et_i[T]}\ ) \wedge \\ (false \Rightarrow U_b^1)\ \wedge\ (V_b^k \Rightarrow E_{l_{break}}) \wedge \\ (\forall et_i = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset) \wedge (et_i \subseteq et)) \end{array} \right) \end{array} \right)$$

$$\Rightarrow$$

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

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:

$$
\begin{array}{l}
\vdash T_{S_1} \quad \wedge \\
[(I_{l_b} ... I_{l_{b\_end}}), et_1] = \\
\quad \nabla_S \, (T_{S_1},\ l_b,\ l_c,\ l_{d+1},\ f,\ et\ ) \wedge \\
(\forall\ T : \mathit{Type} : ((T \preceq \mathit{Throwable}) \vee (T \equiv \mathit{any})) : \\
\qquad\qquad (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{et[l_b, l_{b\_end}, T]}) \quad \wedge \\
(I \quad \Rightarrow \quad E_{l_c}) \ \wedge \\
(Q_b\ \Rightarrow (f = \emptyset\ \Rightarrow\ (Q_b\ \Rightarrow\ E_{l_{d+1}})) \wedge \\
\qquad \left(
\begin{array}{l}
f \neq \emptyset \Rightarrow \forall i\ \in\ 1..k : \\
\left(
\begin{array}{l}
(\vdash \{U^i\}\ s_i\ \{V^i\} \quad \wedge\ \ (V_b^i \Rightarrow U_b^{i+1}) \wedge \\
(\ \forall T : \mathit{Type} : ((T \preceq \mathit{Throwable}) \vee (T \equiv \mathit{any})) : \\
\qquad (V_e^i\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{et_i[T]}\ ) \wedge \\
(Q_b \Rightarrow U_b^1) \ \wedge\ (V_b^k \Rightarrow E_{l_{break}}) \wedge \\
(\forall et_i = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset) \wedge (et_i \subseteq et))
\end{array}
\right)
\end{array}
\right) \\
\Rightarrow \\
\forall\ l\ \in\ l_b\ ...\ l_{b\_end} : \vdash \{E_l\}\ I_l
\end{array}
$$

Since the exception table et is the same than the theorem hypothesis, we know:

$$
\begin{array}{l}
(\forall\ T : \mathit{Type} : ((T \preceq \mathit{Throwable}) \vee (T \equiv \mathit{any})) : \\
\qquad\qquad (R_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV) \ \Rightarrow\ E_{et[l_b, l_{b\_end}, T]})
\end{array}
$$

Since the *finally function* f is $\emptyset$, to be able to apply the induction hypothesis we have to show that

$$Q_b \ \Rightarrow\ E_{l_{d+1}} \tag{D.16}$$

$$I \Rightarrow E_{l_c} \tag{D.17}$$

From the theorem hypothesis we have $((I\ \wedge \neg e)\ \vee\ Q_b\ ) \ \Rightarrow\ E_{l_{d+1}}$. Then $Q_b\ \Rightarrow\ E_{l_{d+1}}$ holds because $Q_b \Rightarrow ((I_n\ \wedge \neg e)\ \vee\ Q_b\ ) \ \Rightarrow\ E_{l_{d+1}}$. The second implication holds since the precondition E_{l_c} is I. Now, we can apply the induction hypothesis and we get:

$$\forall\ l\ \in\ l_b\ ...\ l_{b\_end} : \vdash \{E_l\}\ I_l$$

Thus, we prove:

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

□

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:

$$T_{S_1} \equiv \frac{Tree_1}{\{\ P\ \}\quad s_1\quad \{\ Q_n, Q_b, Q_e\ \}} \qquad T_{S_2} \equiv \frac{Tree_2}{\{\ Q\ \}\quad s_2\quad \{\ R, R'_b, R'_e\ \}}$$

where

$$Q \equiv \left(\begin{array}{l} (Q_n \wedge \mathcal{X}Tmp = normal)\ \vee (Q_b \wedge \mathcal{X}Tmp = break) \vee \\ (\ Q_e[eTmp/excV] \wedge \mathcal{X}Tmp = exc \wedge eTmp = excV\ ) \end{array} \right)$$

$$R \equiv \left(\begin{array}{l} (R'_n \wedge \mathcal{X}Tmp = normal)\ \vee\ (R'_b \wedge \mathcal{X}Tmp = break) \vee \\ (R'_e[eTmp/excV] \wedge \mathcal{X}Tmp = exc) \end{array} \right)$$

We have to prove:

$$
\begin{array}{l}
\vdash \dfrac{T_{S_1} \qquad T_{S_2}}{\left\{\ P\ \right\}\ \texttt{try}\ s_1\ \texttt{finally}\ s_2\ \left\{\ R'_n, R'_b, R'_e\ \right\}} \quad \wedge \\
\left[(I_{l_a} ... I_{l_g}),\ et_4\right] = \\
\nabla_S \left(\dfrac{T_{S_1} \qquad T_{S_2}}{\left\{\ P\ \right\}\ \texttt{try}\ s_1\ \texttt{finally}\ s_2\ \left\{\ R'_n, R'_b, R'_e\ \right\}},\ l_a,\ l_{g+1},\ l_{break},\ f, et \right) \quad \wedge \\
(\forall\ T : \mathit{Type} : ((T \preceq \mathit{Throwable}) \vee (T \equiv \mathit{any})) : \\
\qquad (R'_e\ \wedge\ \mathit{excV} \neq \mathit{null}\ \wedge\ s(0) = \mathit{excV})\ \ \Rightarrow\ \ E_{et[l_a, l_g, T]}) \quad \wedge \\
\left(R'_n \ \ \Rightarrow\ \ E_{l_{g+1}}\right)\ \wedge \\
(R'_b\ \Rightarrow (f = \emptyset\ \Rightarrow\ (R'_b\ \Rightarrow\ E_{l_{break}}))\ \wedge \\
\quad \left(\begin{array}{l}
f \neq \emptyset \Rightarrow \forall i\ \in\ 1..k : \\
\left(\begin{array}{l}
(\vdash \{U^i\}\ s_i\ \{V^i\} \quad \wedge\ \ (V^i_b \Rightarrow U^{i+1}_b)\ \wedge \\
(\ \forall T : \mathit{Type} : ((T \preceq \mathit{Throwable}) \vee (T \equiv \mathit{any})) : \\
\qquad (V^i_e\ \wedge\ \mathit{excV} \neq \mathit{null}\ \wedge\ s(0) = \mathit{excV})\ \ \Rightarrow\ \ E_{et_i[T]}\ )\ \wedge \\
(R'_b \Rightarrow U^1_b)\ \wedge\ (V^k_b \Rightarrow E_{l_{break}})\ \wedge \\
(\forall et_i = r^1_i + r^2_i + ... + r^m_i : (r^1_i \cap r^2_i \cap ... \cap r^m_i = \emptyset) \wedge (et_i \subseteq et))
\end{array} \right)
\end{array} \right) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_g : \vdash \{E_l\}\ I_l
\end{array}
$$

Let f' be a list of *Finally* defined as:

$$
f' = \left[\dfrac{\mathit{Tree}_2}{\left\{\ Q\ \right\}\ \ s_2\ \ \left\{\ R, R'_b, R'_e\ \right\}},\ \mathit{getExcLines}(l_a, l_b, et) \right] + f
$$

Let et' be an exception table defined as:

$$
et' = et + [l_a, l_b, l_d, \mathit{any}]
$$

By the induction hypothesis, we get:

$$
\begin{array}{l}
\vdash T_{S_1} \quad \wedge \\
[(I_{l_a} ... I_{l_{a\_end}}),\ et_2] = \nabla_S\,(T_{S_1},\ l_a,\ l_b,\ l_{break},\ f', et'\ )\ \wedge \\
(\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (Q_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \ \Rightarrow\ \ E_{et'[l_a,l_b,T]})\ \ \wedge \\
(Q_n\ \ \Rightarrow\ \ E_{l_b})\ \wedge \\
(Q_b\ \Rightarrow (f' = \emptyset\ \Rightarrow\ (Q_b\ \Rightarrow\ E_{l_{break}}))\ \wedge \\
\quad \left(\begin{array}{l}
f' \neq \emptyset \Rightarrow \forall i\ \in\ 1..k : \\
\left(\begin{array}{l}
(\vdash \{U^i\}\ s_i\ \{V^i\} \quad \wedge\ \ (V_b^i \Rightarrow U_b^{i+1})\ \wedge \\
(\ \forall T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad (V_e^i\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \ \Rightarrow\ \ E_{et_i''[T]}\ )\ \wedge \\
(Q_b \Rightarrow U_b^1)\ \wedge\ (V_b^k \Rightarrow E_{l_{break}})\ \wedge \\
(\forall et_i'' = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset) \wedge (et_i'' \subseteq et'))
\end{array} \right)
\end{array} \right) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_{a\_end} : \vdash \{E_l\}\ I_l
\end{array}
$$

We have $\vdash T_{S_1}$. Since $et' = et + [l_a, l_b, l_d, any]$, then we can prove:

$$
\begin{array}{l}
\forall\ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (Q_e\ \wedge\ excV \neq null\ \wedge\ s(0) = excV)\ \ \Rightarrow\ \ E_{et'[l_a,l_b,T]}
\end{array}
$$

from the hypothesis (except for the type *any*). In the case of the type is *any*, we have to prove:

$$
\left(\begin{array}{l} Q_e\ \wedge\ \tau(excV) \preceq T\ \wedge \\ excV \neq null\ \wedge\ s(0) = excV) \end{array} \right) \Rightarrow\ E_{l_d}
$$

since the target label of $et'[l_a, l_b, T]$ is l_d. By definition of E_{l_d}, we have

$$
\left(\begin{array}{l} Q_e\ \wedge\ \tau(excV) \preceq T\ \wedge \\ excV \neq null\ \wedge\ s(0) = excV) \end{array} \right) \Rightarrow \left(\begin{array}{l} shift(Q_e)[eTmp/excV]\ \wedge \\ \wedge\ eTmp = excV \end{array} \right)
$$

The implication holds because Q_e does not refer to the stack, and then $shift(Q_e) = Q_e$. Finally, to be able to apply the first induction hypothesis, we need to prove:

$$
\begin{array}{l}
(Q_b \;\Rightarrow (f' = \emptyset \;\Rightarrow\; (Q_b \;\Rightarrow\; E_{l_{break}})) \;\wedge \\
\left(\begin{array}{l}
f' \neq \emptyset \Rightarrow \forall i \;\in\; 1..k : \\
\left(\begin{array}{l}
(\vdash \{U^i\}\; s_i \; \{V^i\} \quad \wedge \;\; (V_b^i \Rightarrow U_b^{i+1}) \;\wedge \\
(\; \forall T : \mathit{Type} : ((T \preceq \mathit{Throwable}) \vee (T \equiv \mathit{any})) : \\
\qquad (\, V_e^i \;\wedge\; \mathit{excV} \neq \mathit{null} \;\wedge\; s(0) = \mathit{excV}) \quad\Rightarrow\quad E_{et_i''[T]} \;) \;\wedge \\
(Q_b \Rightarrow U_b^1) \;\wedge\; (V_b^k \Rightarrow E_{l_{break}}) \;\wedge \\
(\forall et_i'' = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset) \wedge (et_i'' \subseteq et'))
\end{array} \right)
\end{array} \right)
\end{array}
$$

Let et_i'' be the exception table for the current finally block. The exception table for the last finally block is defined as $et_i'' = \mathit{getExcLines}(l_a, l_b, et)$. Using the definition of f', and the hypothesis, we need to prove:

$$Q_b \;\Rightarrow\; U_b^0 \tag{D.18}$$

$$V_b^k \;\Rightarrow\; E_{l_{break}} \tag{D.19}$$

$$V_b^0 \Rightarrow U_b^1 \tag{D.20}$$

$$\begin{array}{l}(\forall\; T : \mathit{Type} : ((T \preceq \mathit{Throwable}) \vee (T \equiv \mathit{any})) : \\ \qquad (\, V_e^i \;\wedge\; \mathit{excV} \neq \mathit{null} \;\wedge\; s(0) = \mathit{excV}) \quad\Rightarrow\quad E_{et_i''[T]} \;)\end{array} \tag{D.21}$$

$$(\forall\; et_i'' = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset) \;\wedge (et_i'' \subseteq et)) \tag{D.22}$$

These expressions hold as follows:

(D.18): $Q_b \;\Rightarrow\; U_b^0$ holds since $U_b^0 \;\equiv\; Q_b$.

(D.19): $V_b^k \;\Rightarrow\; E_{l_{break}}$ holds because V_b^k is the same than the theorem hypothesis, and the implication holds in the theorem hypothesis.

(D.20): $V_b^0 \Rightarrow U_b^1$: from theorem hypothesis, we know that $R_b' \Rightarrow\; U_b^1$. The postcondition V_b^0 is equal to R_b' then $V_b^0 \;\Rightarrow\; U_b^1$

(D.21): We have to prove

$$\begin{array}{l}(\forall\; T : \mathit{Type} : ((T \preceq \mathit{Throwable}) \vee (T \equiv \mathit{any})) : \\ \qquad (R_e' \;\wedge\; \mathit{excV} \neq \mathit{null} \;\wedge\; s(0) = \mathit{excV}) \quad\Rightarrow\quad E_{et_i''[T]}\end{array}$$

because $V_b^0 \equiv R_e'$. 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 *getExcLines* because *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:

$$\forall\ l\ \in\ \{l_b, ...l_{b\_end}\} : \vdash \{E_l\}\ I_l$$

The instructions specifications at l_c, ... l_g holds by definition of *wp*. Joining the proofs, we get

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

□

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:

$$
\begin{array}{l}
\vdash \dfrac{}{\left\{\ P\ \right\}\ \texttt{break}\ \left\{\ \mathit{false}, P, \mathit{false}\ \right\}} \quad \wedge \\
[(I_{l_a}...I_{l_b}),\ et_k] = \nabla_S \left(\dfrac{}{\left\{\ P\ \right\}\ \texttt{break}\ \left\{\ \mathit{false}, P, \mathit{false}\ \right\}},\ l_a, l_{b+1},\ l_{break},\ f, et'_0 \right) \wedge \\
(\forall\ T : \mathit{Type} : ((T \preceq \mathit{Throwable}) \vee (T \equiv \mathit{any})) : \\
\qquad (\mathit{false}\ \wedge\ \mathit{excV} \neq \mathit{null}\ \wedge\ s(0) = \mathit{excV})\ \ \Rightarrow\ \ E_{et'_0[l_a, l_b, T]})\ \ \wedge \\
\big(\mathit{false}\ \ \Rightarrow\ \ E_{l_{b+1}}\big) \wedge \\
(P\ \Rightarrow (f = \emptyset\ \Rightarrow\ (P\ \Rightarrow\ E_{l_{break}})) \wedge \\
\quad \left(\begin{array}{l}
f \neq \emptyset \Rightarrow \forall i\ \in\ 1..k : \\
\left(\begin{array}{l}
(\vdash \{U^i\}\ s_i\ \{V^i\} \quad \wedge\ \ (V_b^i \Rightarrow U_b^{i+1}) \wedge \\
(\ \forall T : \mathit{Type} : ((T \preceq \mathit{Throwable}) \vee (T \equiv \mathit{any})) : \\
\qquad (\ V_e^i\ \wedge\ \mathit{excV} \neq \mathit{null}\ \wedge\ s(0) = \mathit{excV})\ \ \Rightarrow\ \ E_{et_i[T]}\ ) \wedge \\
(P \Rightarrow U_b^1)\ \wedge\ (V_b^k \Rightarrow E_{l_{break}}) \wedge \\
(\forall et_i = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset) \wedge (et_i \subseteq et'_0))
\end{array} \right)
\end{array} \right) \\
\Rightarrow \\
\forall\ l\ \in\ l_a\ ...\ l_b : \vdash \{E_l\}\ I_l
\end{array}
$$

The translation of the break, first adds the translation of all finally blocks f_i, and then it adds a br instruction. Let $f_i = [T_{F_i}, et'_i]$ denote the i-th element of the list f, where

$$T_{F_i} = \frac{\mathit{Tree}_i}{\{\ U^i\ \}\ \ s_i\ \ \{\ V^i\ \}}$$

and U^i and V^i have the following form, which corresponds to the Hoare rule for `try-finally`:

$$U^i \equiv \left\{ \begin{array}{l} (U_n^i \ \wedge \ \mathcal{X}Tmp = normal) \ \vee \ (U_b^i \ \wedge \ \mathcal{X}Tmp = break) \ \vee \\ (\ U_e^i[eTmp/excV] \ \wedge \ \mathcal{X}Tmp = exc \ \wedge \ eTmp = excV \ ) \end{array} \right\}$$

$$V^i \equiv \left\{ \left(\begin{array}{l} (V_n'^i \ \wedge \ \mathcal{X}Tmp = normal) \vee \\ (V_b'^i \ \wedge \ \mathcal{X}Tmp = break) \vee \\ (V_e'^i \ \wedge \ \mathcal{X}Tmp = exc) \end{array} \right), \ V_b^i, \ V_e^i \right\}$$

We can prove that the translation of f_i by induction hypothesis. We show that the hypotheses holds for f_i arbitrary:

$$[b_i, et_i'] = \nabla_S \left(\ T_{F_i}, \ l_{a_i}, l_{a_{i+1}}, \ l_{br}, \ f_{i+1} + ... + f_k, \ divide(et_{i-1}', et_i, l_{a_i}, l_{a_i+1}) \right)$$

Let $et = divide(et_{i-1}', et_i, l_{a_i}, l_{a_i+1})$. We have to prove:

$$\begin{array}{l}
\vdash T_{F_i} \quad \wedge \\
\left[(I_{l_{fi}} ... I_{l_{fi\_end}}), \ et_k\right] = \nabla_S \left(T_{F_i}, \ l_{fi}, l_{fi'}, \ l_{break}, \ f_{i+1} ... f_k, et \ \right) \ \wedge \\
(\forall \ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (V_e^i \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \quad \Rightarrow \quad E_{et[l_{fi}, l_{fi\_end}, T]}) \quad \wedge \\
\left(V_b^i \quad \Rightarrow \quad E_{l_{fi'}} \right) \ \wedge \\
(V_b^i \ \Rightarrow (f = \emptyset \ \Rightarrow \ (V_b^i \ \Rightarrow \ E_{l_{break}})) \ \wedge \\
\qquad \left(\begin{array}{l} f \neq \emptyset \Rightarrow \forall i \ \in \ 1..k : \\ \left(\begin{array}{l} (\vdash \{U^i\} \ s_i \ \{V^i\} \quad \wedge \quad (V_b^i \Rightarrow U_b^{i+1}) \ \wedge \\ (\ \forall T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\ \qquad (V_e^i \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \quad \Rightarrow \quad E_{et_i[T]} \ ) \ \wedge \\ (V_b^i \Rightarrow U_b^1) \ \wedge \ (V_b^k \Rightarrow E_{l_{break}}) \ \wedge \\ (\forall et_i = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset) \wedge (et_i \subseteq et)) \end{array} \right) \end{array} \right) \\
\Rightarrow \\
\forall \ l \ \in \ l_{fi} \ ... \ l_{fi\_end} : \vdash \{E_l\} \ I_l
\end{array}$$

We can prove:

$$\begin{array}{l}
(\forall \ T : Type : ((T \preceq Throwable) \vee (T \equiv any)) : \\
\qquad\qquad (V_e^i \ \wedge \ excV \neq null \ \wedge \ s(0) = excV) \quad \Rightarrow \quad E_{et[l_{fi}, l_{fi\_end}, T]})
\end{array}$$

using the hypothesis (case $f \neq \emptyset$):

$$\begin{array}{l}(\forall\ T : \mathit{Type} : ((T \preceq \mathit{Throwable}) \vee (T \equiv \mathit{any})) : \\ \qquad\qquad (\ V_e^i\ \wedge\ \mathit{excV} \neq \mathit{null}\ \wedge\ s(0) = \mathit{excV})\ \ \Rightarrow\ \ E_{et_i[T]}\ )\end{array}$$

and Lemma 4 that says $et[l_{fi}, l_{fi\_end}, T] = et_i[T]$.

Showing that $V_n^i \Rightarrow E_{lnext}$ holds is equivalent to prove $V_n^i \Rightarrow E_{l_{fi+1}}$. The precondition $E_{l_{fi+1}}$ is defined as U^{i+1}. Then, we have to prove $V_n^i \Rightarrow U^{i+1}$. This implication holds from the hypothesis ($V_b^i \Rightarrow U^{i+1}$ and $V_b^i \Rightarrow V_n^i$).

We can prove the finally proof $f_i ... f_k$

$$\begin{array}{l}(\ V_b^i\ \Rightarrow (f = \emptyset\ \Rightarrow\ (\ V_b^i\ \Rightarrow\ E_{l_{break}}))\ \wedge \\ \quad \left(\begin{array}{l} f \neq \emptyset \Rightarrow \forall i\ \in\ 1..k : \\ \left(\begin{array}{l} (\vdash \{U^i\}\ s_i\ \{V^i\} \quad \wedge\ \ (\ V_b^i \Rightarrow U_b^{i+1})\ \wedge \\ (\ \forall T : \mathit{Type} : ((T \preceq \mathit{Throwable}) \vee (T \equiv \mathit{any})) : \\ \qquad (\ V_e^i\ \wedge\ \mathit{excV} \neq \mathit{null}\ \wedge\ s(0) = \mathit{excV})\ \ \Rightarrow\ \ E_{et_i[T]}\ )\ \wedge \\ (\ V_b^i \Rightarrow U_b^1)\ \wedge\ (\ V_b^k \Rightarrow E_{l_{break}})\ \wedge \\ (\forall et_i = r_i^1 + r_i^2 + ... + r_i^m : (r_i^1 \cap r_i^2 \cap ... \cap r_i^m = \emptyset) \wedge (et_i \subseteq et)) \end{array} \right) \end{array} \right)\end{array}$$

by hypothesis. The translation changes the list f by $f_i ... f_k$. But this property holds for $f_1 ... f_k$.

Now we can apply the induction hypothesis and get $\forall\ l\ \in\ l_{fi}\ ...\ l_{fi\_end} : \vdash \{E_l\}\ I_l$

Finally, we have to prove that:

$$\begin{array}{l} P\ \Rightarrow\ U_b^i\ \mathit{and} \\ \mathit{for\ all}\ i :\ V_b^i\ \Rightarrow\ U_b^{i+1}\ \ \mathit{and} \\ V_b^k\ \Rightarrow\ E_{l_{break}} \end{array}$$

These implications hold by hypothesis.

□

Bibliography

[1] M. Abadi and K. Rustan M. Leino. A logic of object-oriented programs. In *Theory and Practice of Software Development (TAPSOFT)*, volume 1214. Lectures Notes in Computer Science, 1997.

[2] A. Appel. Foundational Proof-Carrying Code. In *LICS '01: Proceedings of the 16th Annual IEEE Symposium on Logic in Computer Science*. IEEE Computer Society, 2001.

[3] K. R. Apt. Ten Years of Hoare's Logic: A Survey—Part i. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.

[4] A. Banerjee, D. Naumann, and S. Rosenberg. Regional Logic for Local Reasoning about Global Invariants. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 5142 of *Lecture Notes in Computer Science*, pages 387–411. Springer-Verlag, 2008.

[5] F. Y. Bannwart and P. Müller. A Logic for Bytecode. Technical Report 469, ETH Zurich, 2004.

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

[7] M. Barnett, R. Deline, M. Fähndrich, R. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology (JOT)*, 3(6):27–56, 2003.

[8] M. Barnett, R. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

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

[10] G. Barthe, B. Grégoire, and M. Pavlova. Preservation of proof obligations from java to the java virtual machine. In *IJCAR '08: Proceedings of the 4th international joint conference on Automated Reasoning*, pages 83–99. Springer-Verlag, 2008.

[11] G. Barthe and C. Kunz. Certificate translation for specification-preserving advices. In *FOAL '08: Proceedings of the 7th workshop on Foundations of aspect-oriented languages*, pages 9–18. ACM, 2008.

[12] G. Barthe and C. Kunz. Certificate translation in abstract interpretation. In *European Symposium on Programming*, pages 368–382, 2008.

[13] G. Barthe, C. Kunz, D. Pichardie, and J. Samborski-Forlese. Preservation of proof obligations for hybrid verification methods. In *Software Engineering and Formal Methods*. IEEE Press, 2008.

[14] G. Barthe, C. Kunz, and J. Sacchini. Certified reasoning in memory hierarchies. In G. Ramalingam, editor, *Asian Programming Languages and Systems Symposium*, Lecture Notes in Computer Science. Springer-Verlag, 2008.

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

[16] N. Benton. A typed logic for stacks and jumps, 2004.

[17] J. Berdine, C. Calcagno, and P. W. Ohearn. Smallfoot: Modular automatic assertion checking with separation logic. In *In International Symposium on Formal Methods for Components and Objects*, pages 115–137. Springer, 2005.

[18] M. Berger, K. Honda, and N. Yoshida. A logical analysis of aliasing in imperative higher-order functions. In *ICFP '05: Proceedings of the tenth ACM SIGPLAN international conference on Functional programming*, pages 280–293. ACM, 2005.

[19] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. Springer-Verlag, 2004.

[20] J. Bicarregui, C. A. R. Hoare, and J. C. P. Woodcock. The verified software repository: a step towards the verifying compiler. *Formal Asp. Comput.*, 18(2):143–151, 2006.

[21] B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ToPLAS*, 2008. To appear.

[22] S. Blazy, Z. Dargaye, and X. Leroy. Formal Verification of a C Compiler Front-End. In *FM 2006: Formal Methods*, pages 460–475, 2006.

[23] J. O. Blech and B. Grégoire. Certifying Code Generation with Coq. In *Proceedings of the 7th Workshop on Compiler Optimization meets Compiler Verification (COCV 2008)*, ENTCS, April 2008.

[24] J. O. Blech and A. Poetzsch-Heffter. A Certifying Code Generation Phase. In *Proceedings of the 6th Workshop on Compiler Optimization meets Compiler Verification (COCV 2007)*, ENTCS, March 2007.

[25] E. Börger, N. G. Fruja, V. Gervasi, and R. F. Stärk. A high-level modular definition of the semantics of C#. *Theoretical computer science*, 336:235–284, 2005.

[26] C. Calcagno, P. O'Hearn, and H. Yang. Local action and abstract separation logic. In *LICS '07: Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science*, pages 366–378, 2007.

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

[28] D. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, volume 37, pages 292–310. ACM Press, November 2002.

[29] C. Colby, P. L. G. Necula, F. Blau, M. Plesko, and K. Cline. A certifying compiler for java. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 95–107. ACM, 2000.

[30] S. A. Cook. Soundness and Completeness of an Axiom System for Program Verification. *SIAM Journal on Computing*, 7(1):70–90, 1978.

[31] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 238–252. ACM, 1977.

[32] Á. Darvas. *Reasoning About Data Abstraction in Contract Languages.* PhD thesis, ETH Zurich, Switzerland, 2009. To appear.

[33] A. Darvas and K. R. M. Leino. Practical reasoning about invocations and implementations of pure methods. In M. B. Dwyer and A. Lopes, editors, *FASE*, volume 4422 of *LNCS*, pages 336–351. Springer-Verlag, 2007.

[34] F. S. de Boer and C. Pierik. How to Cook a Complete Hoare Logic for Your Pet OO Language. In *Formal Methods for Components and Objects*, Lecture Notes in Computer Science, 2004.

[35] E. W. Dijkstra. *A Discipline of Programming.* Prentice Hall PTR, 1997.

[36] D. Distefano and M. J. Parkinson. jStar: Towards Practical Verification for Java. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pages 213–226, 2008.

[37] Y. Dong, S. Wang, L. Zhang, and P. Yang. Modular certification of low-level intermediate representation programs. *33rd Annual IEEE International Computer Software and Applications Conference*, 1:563–570, 2009.

[38] EVE: Eiffel Verification Environment. `http://eve.origo.ethz.ch`.

[39] R. B. Findler and M. Felleisen. Contracts for higher-order functions. *ACM SIGPLAN Notices*, 37(9):48–59, 2002.

[40] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison Wesley, 1994.

[41] G. Goos and W. Zimmermann. Verification of Compilers. In *Correct System Design, Recent Insight and Advances, (to Hans Langmaack on the occasion of his retirement from his professorship at the University of Kiel)*, pages 201–230. Springer-Verlag, 1999.

[42] G. Gorelick. A Complete Axiomatic System for Proving Assertions about Recursive and Non-Recursive Programs. Technical Report TR-75, Department of Computer Science, University of Toronto, 1975.

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

[44] M. Guex. Implementing a Proof-Transforming Compiler from Eiffel to CIL. Technical report, ETH Zurich, 2006.

[45] B. Hauser. Embedding Proof-Carrying Components into Isabelle. Master's thesis, ETH Zurich, 2009.

[46] U. Hensel, M. Huisman, B. Jacobs, and H. Tews. Reasoning about classes in object-oriented languages: Logical models and tools. In *European Symposium on Programming (ESOP)*, volume 1381 of *Lecture Notes in Computer Science*, pages 105–121. Springer-Verlag, 1998.

[47] M. Hess. Integrating Proof-Transforming Compilation into EiffelStudio. Master's thesis, ETH Zurich, 2008.

[48] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of ACM*, 12(10):576–580, 1969.

[49] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*, pages 102–116. Springer-Verlag, 1971.

[50] C. A. R. Hoare. An axiomatic definition of the programming language PASCAL. In *Proceedings of the International Symposium on Theoretical Programming*, pages 1–16. Springer-Verlag, 1974.

[51] K. Honda, N. Yoshida, and M. Berger. An observationally complete program logic for imperative higher-order frame rules. In *LICS '05: Proceedings of the 20th Annual IEEE Symposium on Logic in Computer Science*, pages 260–279. IEEE Computer Society, 2005.

[52] M. Huisman and B. Jacobs. Java program verification via a Hoare logic with abrupt termination. In E. Maibaum, editor, *Approaches to Software Engineering*, volume 1783 of *Lecture Notes in Computer Science*. Springer-Verlag, 2000.

[53] M. Huisman, B. Jacobs, and J. van den Berg. A case study in class library verification: Java's vector class. In *Object-Oriented Technology: ECOOP'99 Workshop Reader*, volume 1743, pages 109–110. Springer-Verlag, 1999.

[54] B. Jacobs. *A Statically Verifiable Programming Model for Concurrent Object-Oriented Programs*. PhD thesis, Katholieke Universiteit Leuven, 2007.

[55] C. B. Jones, P. W. O'Hearn, and J. Woodcock. Verified software: A grand challenge. *IEEE Computer*, 39(4):93–95, 2006.

[56] K. Rustan M. Leino. This is boogie 2. Technical Report Manuscript KRML 178, Microsoft Research, 2008.

[57] K. Rustan M. Leino and J. L. A. van de Snepscheut. Semantics of exceptions. In *PROCOMET*, pages 447–466, 1994.

[58] H. Karahan. Proof-Transforming Compilation of Eiffel Contracts. Technical report, ETH Zurich, 2008.

[59] I. T. Kassios. Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions. In *FM 2006: Formal Methods*, pages 268–283, 2006.

[60] N. Krishnaswami, J. Aldrich, and L. Birkedal. Modular verification of the subject-observer pattern via higher-order separation logic. In *Formal Techniques for Java-like Programs*, 2007.

[61] C. Kunz. *Certificate Translation alongside Program Transformations.* PhD thesis, ParisTech, 2009.

[62] G. T. Leavens, A. L. Baker, and C. Ruby. JML: A Notation for Detailed Design, 1999.

[63] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.

[64] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual, 2006.

[65] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer-Verlag, 2004.

[66] K. R. M. Leino and P. Müller. Modular verification of static class invariants. In J. Fitzgerald, I. Hayes, and A. Tarlecki, editors, *Formal Methods (FM)*, volume 3582 of *Lecture Notes in Computer Science*, pages 26–42. Springer-Verlag, 2005.

[67] K. R. M. Leino and P. Müller. Verification of equivalent-results methods. In *European Symposium on Programming (ESOP)*, volume 4960 of *Lecture Notes in Computer Science*, pages 307–321. Springer-Verlag, 2008.

[68] X. Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. *POPL '06: Proceedings of the 33nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 41(1):42–54, 2006.

[69] H. Liu and J. S. Moore. Java program verification via a jvm deep embedding in acl2. In *17th International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pages 184–200, 2004.

[70] C. Luo, G. He, and S. Qin. A heap model for java bytecode to support separation logic. *Asia-Pacific Software Engineering Conference*, 0:127–134, 2008.

[71] B. Meyer. Design by contract. In D. Mandrioli and B. Meyer, editors, *Advances in Object-Oriented Software Engineering*, pages 1–50. Prentice Hall, 1991.

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

[73] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, second edition, 1997.

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

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

[76] MOBIUS Consortium. Deliverable 3.1: Byte code level specification language and program logic. Available online from `http://mobius.inria.fr`, 2006.

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

[78] MOBIUS Consortium. Deliverable 4.5: Report on proof transformation for optimizing compilers. Available online from `http://mobius.inria.fr`, 2008.

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

[80] P. Müller and M. Nordio. Proof-Transforming Compilation of Programs with Abrupt Termination. In *SAVCBS '07: Proceedings of the 2007 conference on Specification and verification of component-based systems*, pages 39–46, 2007.

[81] P. Müller and M. Nordio. Proof-Transforming Compilation of Programs with Abrupt Termination. Technical Report 565, ETH Zurich, 2007.

[82] P. Müller and A. Rudich. Ownership transfer in Universe Types. In *Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 461–478. ACM, 2007.

[83] P. Müller and J. N. Ruskiewicz. A modular verification methodology for C# delegates. In U. Glässer and J.-R. Abrial, editors, *Rigorous Methods for Software Construction and Analysis*, To appear.

[84] G. Necula. Proof-Carrying Code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, 1997.

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

[86] G. Necula. Translation validation for an optimizing compiler. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 35(5):83–94, 2000.

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

[88] G. Necula and P. Lee. Safe, Untrusted Agents Using Proof-Carrying Code. In *Mobile Agents and Security*, pages 61–91. Springer-Verlag, 1998.

[89] T. Nipkow and T. U. Mnchen. Jinja: Towards a Comprehensive Formal Semantics for a Java-like Language. In *In Proceedings of the Marktoberdorf Summer School. NATO Science Series.* Press, 2003.

[90] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

[91] M. Nordio, C. Calcagno, B. Meyer, and P. Müller. Reasoning about Function Objects. Technical Report 615, ETH Zurich, 2009.

[92] M. Nordio, C. Calcagno, B. Meyer, P. Müller, and J. Tschannen. Reasoning about Function Objects. In J. Vitek, editor, *TOOLS-EUROPE*, Lecture Notes in Computer Science. Springer-Verlag, 2010.

[93] M. Nordio, C. Calcagno, P. Müller, and B. Meyer. A Sound and Complete Program Logic for Eiffel. In M. Oriol, editor, *TOOLS-EUROPE 2009*, volume 33 of *Lecture Notes in Business and Information Processing*, pages 195–214, 2009.

[94] M. Nordio, C. Calcagno, P. Müller, and B. Meyer. Soundness and Completeness of a Program Logic for Eiffel. Technical Report 617, ETH Zurich, 2009.

[95] M. Nordio, P. Müller, and B. Meyer. Formalizing Proof-Transforming Compilation of Eiffel programs. Technical Report 587, ETH Zurich, 2008.

[96] M. Nordio, P. Müller, and B. Meyer. Proof-Transforming Compilation of Eiffel Programs. In R. Paige and B. Meyer, editors, *TOOLS-EUROPE*, Lecture Notes in Business and Information Processing. Springer-Verlag, 2008.

[97] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 268–280, 2004.

[98] PACO: A Proof-Transforming Compiler for Eiffel. `http://paco.origo.ethz.ch/`.

[99] R. Paige and J. Ostroff. ERC: an Object-Oriented Refinement Calculus for Eiffel. *Formal Aspects of Computing*, 16:51–79, 2004.

[100] M. J. Parkinson. *Local Reasoning for Java.* PhD thesis, University of Cambridge, 2005.

[101] M. J. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, volume 40, pages 247–258. ACM, 2005.

[102] M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 75–86. ACM, 2008.

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

[104] A. Pnueli, M. Siegel, and E. Singerman. Translation Validation. In *TACAS '98: Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, pages 151–166. Springer-Verlag, 1998.

[105] A. Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs.* PhD thesis, Habilitation thesis, Technical University of Munich, January 1997.

[106] A. Poetzsch-Heffter and M. J. Gawkowski. Towards Proof Generating Compilers. *ENTCS*, 132(1):37–51, 2005.

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

[108] A. Poetzsch-Heffter and P. Müller. A Programming Logic for Sequential Java. In S. D. Swierstra, editor, *European Symposium on Programming Languages and Systems (ESOP'99)*, volume 1576 of *LNCS*, pages 162–176. Springer-Verlag, 1999.

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

[110] C. L. Quigley. A programming logic for java bytecode programs. In *Theorem Proving in Higher Order Logics*, pages 41–54. Springer, 2003.

[111] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[112] A. Saabas. *Logics for Low-Level Code and Proof-Preserving Program Optimizations.* PhD thesis, Tallinn University of Technology, 2008.

[113] A. Saabas and T. Uustalu. Proof optimization for partial redundancy elimination. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation*, pages 91–101. ACM Press, 2008.

[114] A. Saabas and T. Uustalu. Proof optimization for partial redundancy elimination. *Journal of Logic and Algebraic Programming*, 2009. To appear.

[115] N. Schirmer. A verification environment for sequential imperative programs in Isabelle/HOL. In F. Baader and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 3452 of *LNAI*, pages 398–414. Springer, 2005.

[116] N. Schirmer. *Verification of Sequential Imperative Programs in Isabelle/HOL.* PhD thesis, Technische Universität München, 2006.

[117] B. Schoeller. *Making classes provable through contracts, models and frames.* PhD thesis, ETH Zurich, 2007.

[118] J. Smans. *Specification and automatic verification of frame properties for Java-like programs.* PhD thesis, Katholieke Universiteit Leuven, May 2009.

[119] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. In *Formal Techniques for Java-like Programs*, 2008.

[120] J. Smans, B. Jacobs, and F. Piessens. Implicit Dynamic Frames: Combining Dynamic Frames and Separation Logic. In *European Conference on Object-Oriented Programming (ECOOP)*, Lecture Notes in Computer Science. Springer-Verlag, 2009.

[121] J.-B. Tristan and X. Leroy. Formal verification of translation validators: a case study on instruction scheduling optimizations. In *POPL '08: Proceedings of the annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, volume 43, pages 17–27. ACM, 2008.

[122] J. Tschannen. Automatic Verification of Eiffel Programs. Master's thesis, ETH Zurich, 2009.

[123] D. von Oheimb. *Analyzing Java in Isabelle/HOL - Formalization, Type Safety and Hoare Logic -*. PhD thesis, Universität München, 2001.

[124] D. von Oheimb. Hoare Logic for Java in Isabelle/HOL. In *special issue of Concurrency and Computation: Practice and Experience*, volume 13, pages 1173–1214, November 2001.

[125] J. Woodcock. Formal techniques and operational specifications. *Software Engineering Notes*, 14, 1989.

[126] J. Woodcock. First steps in the verified software grand challenge. *IEEE Computer*, 39(10):57–64, 2006.

[127] J. Woodcock and R. Banach. The verification grand challenge. *Computer Society of India Communications*, 31(2)::33–36, 2007.

[128] L. Zuck, A. Pnueli, Y. Fang, and B. Goldberg. VOC: A Methodology for the Translation Validation of Optimizing Compilers. *Journal of Universal Computer Science*, 9(3):223–247, 2003.

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.
- July 2003 - June 2005: Research Assistant at Cordoba Agency of Scientific and Technical Research (Agencia Cordoba Ciencia). Argentina.
- August 2002 - June 2005: Teaching assistant at National University of Río Cuarto. Argentina.