DISS. ETH NO. 30699

FORMALLY VALIDATING TRANSLATIONAL PROGRAM VERIFIERS

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES

(Dr. sc. ETH Zurich)

presented by

GAURAV PARTHASARATHY

MSc ETH CS, ETH Zurich

born on 24.04.1992

accepted on the recommendation of

Prof. Dr. Peter Müller, examiner

Prof. Dr. Alexander J. Summers, co-examiner

Dr. Jean-Christophe Filliâtre, co-examiner

2024

Abstract

Automated program verifiers automatically check whether a software program conforms to a user-provided specification. This includes checking whether the program will not crash and whether the program will compute the results specified by the user for every feasible program execution. Such verifiers perform this automatic check by analysing the input program *statically* (*i.e.* without executing the program) using techniques based on foundations in *formal program verification*. For successful verification results to be meaningful, it is crucial that automated program verifiers are *sound*: that is, whenever the verifier reports success, then the input program must indeed conform to the user-provided specification. Unfortunately, there are no formal soundness guarantees for many verifier implementations used in practice, which are themselves complex software programs. Thus, software bugs in these implementations can and do arise, which compromises the trustworthiness of successful verification results.

Translational program verifiers form one large class of automated program verifiers whose implementations typically do not come with formal soundness guarantees. These verifiers apply a series of program-to-program translations before ultimately producing a set of logical formulas, whose validity is automatically discharged via external tools such as SMT solvers. Such verifiers are sound only if the validity of the produced logical formulas implies that the input program conforms to its specification. In this dissertation, we develop techniques with which one can establish this soundness requirement for translational program verifier implementations used in practice. In particular, one need not implement a verifier from scratch to use our techniques: our techniques can be applied to *existing* implementations used in practice. Given a formal semantics of the input program, our techniques use a *formal per-run validation* approach: for each run of the verifier implementation, we *automatically* generate a *certificate*, which formally proves the soundness requirement. Our generated certificates are expressed in terms of a formal operational semantics of the input program. Moreover, these certificates are expressed in an interactive theorem prover (Isabelle, in our case), and thus can be automatically checked in a trustworthy way.

We have applied our techniques to two existing verifier implementations used in practice: (1) the Boogie verifier implementation, which applies a series of complex Boogie-to-Boogie transformations before ultimately producing logical conditions, and (2) the Viper verifier implementation, which translates Viper programs to Boogie programs. As a result of our work, both of these verifier implementations are able to automatically generate certificates. Our techniques are designed in a general way such that they could be adapted to other translational program verifiers. In particular, our technique for the Viper verifier is designed more generally for translations *into* intermediate verification languages such as Boogie, which form the core of translational program verifiers.

Zusammenfassung

Automatisierte Verifikationstools für Softwareprogramme prüfen automatisch, ob ein Programm eine benutzerdefinierte Spezifikation erfüllt. Dieser Prozess prüft unter anderem, ob Ausführungen des Programms nicht abstürzen, und ob das Programm für jede mögliche Ausführung die korrekten Resultate gemäss der benutzerdefinierten Spezifikation berechnet. Solche Verifikationstools realisieren diese automatische Prüfung durch eine *statische* Analyse des Eingabeprogramms (das heisst, ohne das Programm auszuführen). Diese statischen Analysen basieren auf Grundlagen der *formalen Programmverifikation*. Damit erfolgreiche Verifikationsresultate aussagekräftig sind, ist es entscheidend, dass automatisierte Verifikationstools *korrekt* sind: Das heisst, wenn das Verifikationstool ein erfolgreiches Verifikationsresultat meldet, dann muss das Programm tatsächlich die benutzerdefinierte Spezifikation erfüllen. Leider gibt es für viele in der Praxis verwendeten Verifikationstools keine formalen Korrektheitsgarantien. Die Implementierungen dieser Tools sind selber komplexe Softwareprogramme. Daher treten in solchen Implementierungen Softwarefehler auf, was die Vertrauenswürdigkeit erfolgreicher Verifikationsresultate beeinträchtigt.

Übersetzende Verifikationstools bilden eine grosse Klasse von automatisierten Verifikationstools, für welche es in der Regel keine formalen Korrektheitsgarantien gibt. Diese Verifikationstools wenden eine Sequenz von Programm-zu-Programm Übersetzungen an, bevor sie schlussendlich eine Menge von logischen Formeln generieren, und dann die Gültigkeit dieser Formeln automatisch durch externe Tools (wie z.B. SMT Solver) prüfen. Solche Verifikationstools sind nur dann korrekt, wenn die Gültigkeit der generierten Formeln impliziert, dass das Eingabeprogramm die zugehörige Spezifikation erfüllt. In dieser Dissertation entwickeln wir Techniken, mit denen man diese Korrektheitsbedingung für übersetzende Verifikationstools, welche in der Praxis verwendet werden, beweisen kann. Insbesondere muss ein Verifikationstool nicht von Grund auf neu implementiert werden, damit man unsere Techniken anwenden kann: Unsere Techniken können auf *existierende* Implementierungen angewendet werden, die in der Praxis verwendet werden. Unsere Techniken verwenden einen *formalen Validierungsansatz*, welcher jeden Durchlauf des Verifikationstools separat validiert: Für jeden Durchlauf des Verifikationstools generieren wir *automatisch* ein *Zertifikat*, welches die Korrektheitsbedingung formal beweist. Unsere generierten Zertifikate werden mit Hilfe einer formalen Semantik des Eingabeprogramms ausgedrückt; die Existenz einer solchen formalen Semantik ist eine Voraussetzung für unsere Techniken. Ausserdem werden unsere Zertifikate in einem interaktiven Theorembeweiser ausgedrückt (wir verwenden Isabelle). Somit können diese Zertifikate automatisch und vertrauenswürdig überprüft werden.

Wir haben unsere Techniken auf zwei existierende Implementierungen von Verifikationstools angewendet, die in der Praxis verwendet werden: (1) die Implementierung des Boogie Verifikationstools, welche zuerst eine Sequenz von komplexen Boogie-zu-Boogie Transformationen anwendet, und dann vom resultierenden Boogie-Programm logische Formeln generiert, und (2) die Implementierung des Viper Verifikationstools, welche Viper-Programme in Boogie-Programme übersetzt. Als Ergebnis unserer Arbeit können beide Implementierungen automatisch Zertifikate generieren. Unsere Techniken sind allgemein konzipiert worden, so dass man sie auch auf andere Verifikationstools anwenden könnte. Insbesondere ist unsere Technik für das Viper Verifikationstool allgemein für Übersetzungen in für Verifikation entworfene Zwischensprachen wie Boogie, welche ein zentraler Bestandteil von übersetzenden Verifikationstools sind, konzipiert worden.

Acknowledgements

Doing a PhD is much more enjoyable if one has an outstanding supervisor such as Peter Müller. Peter, thank you for always supporting me, for always showing me how to improve as a researcher, and for creating such a wonderful research environment. I am still amazed by your ability to ask the most important questions, which transformed seemingly insurmountable problems into well-defined next steps and into high-level visions clarifying the actual purpose of solving the problems we were facing. You always set the highest standard for everything you do, which helped me raise the quality of my own work significantly; I hope I am able to reach your consistently high bar of excellence eventually. In a field where it is normal to postpone answering requests or questions (or to forget them entirely), you never did; you always took the time to help out with your full attention. Finally, I am deeply grateful that you always put the well-being of people before anything else.

Had I not met Alex Summers, I likely would have never become a PhD student in the first place. Alex, thank you for introducing me to research during my undergraduate and MSc degrees, and for always believing in me and my work. Your meticulous approach to conducting scientific research was one of the main reasons why I chose to do a PhD. I was very fortunate to continue working with you as a second supervisor during my PhD. Your deep scientific curiosity often made me rethink my solutions, pushed my work further, and gave me a more profound understanding of what I was doing. I am also deeply grateful for our non-technical conversations about the PhD in general, which helped me appreciate the work I was doing more.

I thank Jean-Christophe Filliâtre for serving on my doctoral examination committee and for providing valuable comments on this dissertation. Moreover, I would like to thank the following people for providing valuable feedback on my dissertation: Denis Carnier, Thibault Dardinier, Marco Eilers, Michael Sammler, and most importantly my supervisors Peter and Alex.

My PhD experience was deeply impacted by my scientific collaboration with Thibault Dardinier. Thibault, thank you for helping me grow as a researcher, for never giving up on our joint research vision even when things were looking grim, for always keeping your door open for me no matter how busy you were, and for the countless discussions that broadened my perspective on research and life in general. Even though we have been working together for so long now, I am still amazed at your incredibly deep insights, absolutely exceptional problem-solving skills, and your relentless pursuit of discovering the full truth behind scientific problems. It has been an honour working with you.

I had the pleasure of doing two amazing research internships: (1) one immediately before my PhD with Derek Dreyer at MPI-SWS, and (2) one during my PhD with Rustan Leino at AWS. Derek, thank you for taking the time during my internship to answer many of my questions about research, which further convinced me to do a PhD. I am also deeply grateful that you invited me to work on the prophecies paper during my PhD, which increased my confidence in my abilities during a time when things were not yet working out smoothly in my PhD topic. Rustan, I loved our technical meetings on various verification challenges. Thank you for reminding me why verification is such a fun research area and for devoting so much of your time to support me during my internship.

My PhD experience was greatly enriched by being a part of the Programming Methodology (PM) Group. I am thankful to all its members for establishing an excellent research environment. I thank João C. Pereira for many long discussions on life and academia, which inspired me and reminded me why I got into PL in the first place. I thank Felix Wolf for our countless humorous conversations and for doing an excellent job leading EProg TA teams. I thank Marco Eilers for the many fun conversations on academia, and for being one of the first PhD students to show me the different aspects of research. I thank Linard Arquint for demonstrating by example how to do things the right way systematically, Vytautas Astrauskas for creating an inclusive environment and for the Saturday lunches, Aurel Bílý for fun Wednesday discussions, Lea Brugger for the miraculous effort of establishing a PM football (not soccer!) tradition, Alexandra Bugariu for all of the advice on research, Martin Clochard for helpful technical discussions, Xavier Denis for fun discussions on future possibilities in verification and beyond, Jérôme Dohrau for showing me how to improve as a TA, Jonàš Fiala for reintroducing social events into the group, Nick Klose for ensuring a high vibe quality at all times, Anqi Li for inspiring me to be more focused, Hongyi Ling for demonstrating deep mathematical skills that revealed the essence of complex problems, Christoph Matheja for discussions on how to do research, Wytse Oortwijn for discussions on academia, Fábio Pakk Selmi-Dei for supporting the Viper infrastructure, Federico Poli for showing me what professionalism really means, Michael Sammler for bringing the Viper Roots vision to the next level and showing me a new way of doing research effectively, Malte Schwerhoff for sharing his deep and broad experience with Viper, Dionisis Spiliopoulos for recounting his many adventures in hilarious ways, Arshavir Ter-Gabrielyan for always taking the time to show me how things worked, and Yushuo Xiao for bringing vague research ideas to life through solutions that I would not have otherwise had the privilege of witnessing during my PhD. Finally, I would like to thank Sandra Schneider and Marlies Weissert for their outstanding administrative support, which allowed me to focus on my main PhD work.

I was also fortunate to have many interesting and helpful conversations with other researchers. First, I would like to thank the Programming Language Foundations Lab whose members share similar research interests to our group and who enriched our research environment. In particular, I thank Ralf Jung, Isaac van Bakel, Johannes Hostert, Rudy Peterson, and Max Vistrup. Second, I would like to thank other researchers with whom I interacted at ETH and outside ETH during my PhD, including Luca Beurer-Kellner, Benjamin Bichsel, Pavol Bielik, Denis Carnier, Hai Dang, Michael Faes, Son Ho, Vedant Nanda, Clément Pit-Claudel, Damir Pulatov, Shaz Qadeer, Divya Raghunathan, Koustuv Saha, Joshua Schneider, Reza Sefidgar, Gagandeep Singh, Samuel Steffen, Jack Stodart, Aaron Tomb, and Chris Wendler.

As part of my PhD, I helped supervise outstanding BSc and MSc students whose work either directly contributed to this dissertation or indirectly gave me a more profound understanding of aspects of this dissertation: Ellen Arlt, Benjamin Bonneau, Thibault Dardinier, Alain Delaët–Tixeuil, Lukas Himmelreich, Aleksandar Hubanov, Andrew Lee, Hongyi Ling, Anouk Paradis, Noé Weeks, and Yushuo Xiao. Thank you for your excellent work!

I did not understand what mathematics was about until I met my high school ("Kantonsschule") mathematics teacher Armin Barth. I thank Herr Barth for his outstanding mathematics lectures, which gave me the confidence to pursue computer science at university. Moreover, I thank my high school ("Kantonsschule") history teacher Rémy Kauffmann who showed me how impactful it can be to clearly communicate ideas.

Finally, I thank my friends and family for being there for me during this journey. Most importantly, I am deeply grateful to my parents, Usha and Krishnan, for their unconditional support every step along the way. Irrespective of how things were going during my PhD, they were always there for me.

Contents

Introduction 1.

In many cases it is crucial that software behaves as intended. This means, for instance, that software does not crash and always computes the expected results. Cases in which such a property is crucial includes software that deals with financial transactions, works with sensitive information, or more generally provides core aspects of a product used by many users. The most common approach for increasing the confidence of software in such cases is to test its behaviour on a *finite* set of use cases by executing the software on those use cases. Testing cannot take every possible use case into account because there may be too many such that executing the software on all of them would not be achievable in reasonable time. Frequently, there are even infinitely many use cases (for instance, software may depend on parameters that can be instantiated in infinitely many ways).

To gain more confidence, *formal program verification* techniques can be applied in order to check that software behaves as expected for *every* possible use case; this is achieved by *statically* analysing the software. Such a *static* analysis just inspects the software's source code, but does not actually execute the software. The significantly stronger guarantee obtained via formal program verification compared to testing comes at a nontrivial cost. To obtain these guarantees one must apply complex program reasoning techniques such as deriving a formal proof in a *program logic*. Doing so manually is time-consuming and error-prone. For instance, when using a program logic one must choose a sequence of rules in the logic and perform low-level reasoning steps such as proving nontrivial entailments between logical conditions. To decrease this cost, *automated program verifiers* have been developed, which automate the application of program reasoning techniques. More precisely, given an input software program, these verifiers require users to provide a specification for the program (capturing the possible use cases and the expected results) as well as some auxiliary annotations on the program, both of which are expressed via some formal language. Given this information, the verifier automatically checks whether the input program conforms to the provided specification.

For successful verification results to be meaningful, it is crucial that the automated program verifier is *sound*. That is, if the verifier successfully verifies an input program, then the input program indeed conforms to its provided specification. It is *not* sufficient to ensure the soundness of the program logics that verifiers employ: these verifiers are complex software programs themselves, and thus it is essential that formal guarantees also cover their *actual implementations*, where bugs can and do arise. For many automated program verifiers, no formal guarantees are shown for their implementations, potentially raising doubts as to the trustworthiness of successful verification results.

Translational program verifiers form one large class of automated program verifiers, for which typically no formal guarantees are shown. These verifiers apply a series of program-to-program translations to an input program, followed by a reduction of the resulting program to a set of logical formulas called *verification conditions*. The verifier reports verification success if the verification conditions are valid; validity is usually checked using an SMT solver. Examples of translational program verifiers include Boogie [1], Creusot [2], Dafny [3], Frama-C [4], Gobra [5], VerCors [6], and Viper [7].

[1]: Leino (2008), *This is Boogie 2*

[2]: Denis et al. (2022), *Creusot: A Foundry for the Deductive Verification of Rust Programs*

[3]: Leino (2010), *Dafny: An Automatic Program Verifier for Functional Correctness*

[4]: Kirchner et al. (2015), *Frama-C: A software analysis perspective*

[5]: Wolf et al. (2021), *Gobra: Modular Specification and Verification of Go Programs*

[6]: Blom et al. (2017), *The VerCors Tool Set: Verification of Parallel and Concurrent Software*

[7]: Müller et al. (2016), *Viper: A Verification Infrastructure for Permission-Based Reasoning*

In practice, translational program verifiers are implemented in efficient mainstream programming languages, use diverse libraries and programming paradigms, include subtle optimisations, and use approaches that scale to a large subset of the source language. However, prior work on ensuring the soundness of translational program verifiers is based on implementations expressed on paper or in an interactive theorem prover via languages not used for practical implementations. These idealised implementations omit many of the optimisations performed by practical implementations and are typically tailored just to the source language subset for which soundness is proved. There is a very large gap between actual implementations used in practice and the idealised implementations proved sound. This dissertation bridges this gap for the first time, developing and applying techniques that formally establish the soundness of *existing* implementations of translational program verifiers used in practice. We assume that the produced verification conditions are valid if verification is successful; our results can be combined with work on establishing the soundness of SMT solvers [8–10] to obtain end-to-end guarantees. The soundness results produced by this dissertation are expressed in terms of an operational semantics of the source language. Moreover, these results are expressed and automatically checked in an interactive theorem prover (ITP) such as Coq [11], Isabelle [12], and Lean [13]. ITPs are designed to be highly trustworthy, which significantly increases the trustworthiness of the produced soundness results.

[8]: Böhme et al. (2010), *Fast LCF-Style Proof Reconstruction for Z3*

[9]: Ekici et al. (2017), *SMTCoq: A Plug-In for Integrating SMT Solvers into Coq*

[10]: Fleury et al. (2019), *Reconstructing veriT Proofs in Isabelle/HOL*

[11]: The Coq Development Team (2024), *The Coq Reference Manual – Release 8.19.0*

[12]: Nipkow et al. (2002), *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*

[13]: Moura et al. (2015), *The Lean Theorem Prover (System Description)*

One possible approach to obtain trustworthy soundness results for an existing translational program verifier is to import the verifier implementation into an ITP and to then prove soundness of the implementation *once and for all* in the ITP. However, such a once-and-for-all approach is practically infeasible since existing implementations are large and complex, use a variety of libraries, and are typically written in mainstream programming languages which themselves lack a formalisation. Even if one ignores these aspects, convincing developers of verifiers to instead rewrite implementations using the formalised internal languages that are part of ITPs themselves is challenging. These ITP languages typically lack the mature tool infrastructures and the vast libraries provided by mainstream programming languages.

Instead, we develop *translation validation* approaches, which check the soundness of each verifier run separately. In particular, given a formal semantics for the input program, our approaches *automatically* generate a formal *certificate* on every run of the verifier via an instrumentation of the *existing* implementation. Our instrumentation makes only small adjustments to the existing implementation. Therefore, developers need not change their implementation substantially and they retain the benefits of mainstream programming languages. Moreover, our certificates formally establish soundness for a particular verifier run, where soundness is expressed via the formal semantics of the input program. Our certificates are expressed along with the formal semantics of the input language in the Isabelle ITP, and thus provide strong guarantees that can be checked independently. These certificates contain sufficient information such that Isabelle is able to check them *automatically*. Full automation for checking certificates, without requiring any user guidance at all, was an important objective for us because it is crucial in our setting. The main reason is that our generated certificates are nontrivial and can become quite large. As a result, if we did not provide full automation for checking certificates, then this would (1) lead to a large overhead for users, (2) require users to have expert knowledge about details of our certificates, and (3) require users to know how to use tools such as interactive theorem provers. Thus, if we did not provide full automation, then our approach would be much less attractive.

1.1. Translational Program Verifiers: An Overview

The goal of a translational program verifier is to obtain verification conditions (VCs) whose validity implies that the input program conforms to the provided specification. Performing separate program-to-program translations before computing the VCs instead of directly computing VCs reduces the complexity of each translation and modularises the development of the verifier. Moreover, this separation of concerns enables the reuse of infrastructure across *different* translational program verifiers. This is exemplified in practice: many translational program verifiers translate a program to an *intermediate verification language (IVL)*. An IVL comes with its own verifier that ultimately reduces IVL programs to VCs. For a given IVL, we call a translation from a source language (different from the IVL) into the IVL a *front-end translation*, and we call the verifier reducing the IVL to VCs a *back-end verifier*. This translational approach via an IVL allows for the reuse of the IVL's back-end technology across multiple translational program verifiers (*i.e.* different front-end translations can target the *same* IVL), and makes for a more understandable target representation than direct mappings to logical formulas, simplifying the development of state-of-the-art program verifiers. Note that IVL back-end verifiers are sometimes translational verifiers themselves, applying various program-to-program translations before reducing programs to VCs. They may achieve the reduction to VCs via translation to yet another IVL that is then reduced to VCs via its own back-end verifier.

The advantages of IVLs for program verifiers are similar to the advantages of intermediate program representations such as LLVM [14] for compilers. For instance, several common optimisations may be performed by a back-end verifier, and thus these optimisations benefit any front-end translation that translates into the corresponding IVL. This is similar to, for instance, optimisations performed on LLVM programs that benefit any compiler that targets LLVM. Moreover, having an IVL allows supporting different back-end verifiers for the same IVL that use different verification approaches (similar to different compilers for intermediate representations targetting different hardware) without affecting any of the front-end translations.

[14]: Lattner et al. (2004), *LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation*

There are many examples of front-end translations. Corral [15], Dafny [3], and SMACK [16] translate to the imperative Boogie IVL [1]. Creusot [2]

[15]: Lal et al. (2014), *Powering the static driver verifier using corral*

[3]: Leino (2010), *Dafny: An Automatic Program Verifier for Functional Correctness*

[16]: Carter et al. (2016), *SMACK software verification toolchain*

[1]: Leino (2008), *This is Boogie 2*

[2]: Denis et al. (2022), *Creusot: A Foundry for the Deductive Verification of Rust Programs*

[4]: Kirchner et al. (2015), *Frama-C: A software analysis perspective*

[17]: Filliâtre et al. (2013), *Why3 — Where Programs Meet Provers*

[5]: Wolf et al. (2021), *Gobra: Modular Specification and Verification of Go Programs*

[18]: Eilers et al. (2018), *Nagini: A Static Verifier for Python*

[19]: Astrauskas et al. (2019), *Leveraging Rust Types for Modular Specification and Verification*

[6]: Blom et al. (2017), *The VerCors Tool Set: Verification of Parallel and Concurrent Software*

[7]: Müller et al. (2016), *Viper: A Verification Infrastructure for Permission-Based Reasoning*

[20]: Reynolds (2002), *Separation logic: A logic for shared mutable data structures*

and Frama-C [4] translate to the functional Why3 IVL [17]. Gobra [5], Nagini [18], Prusti [19], and VerCors [6] translate to the imperative Viper IVL [7], which supports separation logic [20] reasoning. The back-end verifiers for the Boogie IVL and the Viper IVL are both translational verifiers themselves. (Viper has multiple back-end verifiers, one of which is a translational verifier and one of which uses symbolic execution.) Boogie's verifier applies multiple Boogie-to-Boogie transformations before computing the VCs, while Viper's verifier uses yet another front-end translation: translating Viper programs to Boogie programs.

There is a wide variety of program-to-program translations applied by translational program verifiers that have different purposes such as (1) simplifying the generation of VCs or (2) making the generated VCs more efficient (*e.g.* in terms of space). Examples for the former purpose include (1) front-end translations that translate complex features in one language to simpler features in a target IVL, (2) the elimination of loops via loop invariants to avoid fixed-point computations for the VCs, and (3) the elimination of polymorphism to more easily generate VCs supported by existing solvers. An example for the latter purpose is the elimination of assignments by the introduction of fresh variables (analogous to static single assignment form in compilers) and suitable `assume` commands [21–23].

[21]: Flanagan et al. (2001), *Avoiding exponential explosion: generating compact verification conditions*
[22]: Leino (2005), *Efficient weakest preconditions*
[23]: Barnett et al. (2005), *Weakest-precondition of unstructured programs*

Since program-to-program translations differ significantly, it is natural to split the soundness of a translational verifier into multiple parts: (1) the soundness of each program-to-program translation (if the target program conforms to its specification then so does the source program), (2) the soundness of the reduction from the final program to the VCs (if the VCs are valid, then the final program conforms to its specification), and (3) the soundness of the solver (if the solver reports success for a VC, then the VC is valid). These separate soundness results together ensure the soundness of the translational verifier as a whole. Each of these results has separate challenges. Therefore, it is natural to develop different approaches for proving soundness. This dissertation develops approaches for the soundness of program-to-program translations and for the reduction of the final program to the VCs. Establishing the soundness of solvers is a much better-studied problem, which this dissertation does not tackle. This dissertation's results can be combined with work on establishing the soundness of SMT solvers [8–10] to obtain end-to-end guarantees.

[8]: Böhme et al. (2010), *Fast LCF-Style Proof Reconstruction for Z3*
[9]: Ekici et al. (2017), *SMTCoq: A Plug-In for Integrating SMT Solvers into Coq*
[10]: Fleury et al. (2019), *Reconstructing veriT Proofs in Isabelle/HOL*

1.2. State of the Art

This section presents the state of the art, which is related to the formal validation of translational program verifiers. The presentation is kept at a high level; some works are discussed in more detail in the related work sections of the technical chapters (see Section 2.11 and Section 3.7).

Translational program verifiers

There are various works that formally prove the soundness of front-end translations once and for all, either on paper or in an ITP. These proofs include translations from BSP, C, and Maple into the Why3 IVL [24–26], translations from Java Bytecode and an object-oriented language into Boogie [27, 28], a translation from the Dminor data processing language into the Bemol IVL [29], and a translation from Chalice into the Viper IVL [30]. These formalisations do not reflect typical implementations used in practice, which are implemented in mainstream languages and apply subtle optimisations not present in these formalisations. Therefore, these results validate an idealised implementation of translational program verifiers and the high-level design of the verifier, but do not validate the actual implementations used in practice, which contain many complexities not captured by the formalised results. For instance, Backes et al. [29] do not connect their formalised Coq implementation with their F# implementation that is actually executed. Herms [25] use an executable extracted from their formalised Coq implementation; they discuss an optimisation of the translation, which they have not implemented, but which is included in a more practical verifier implementation.

[24]: Fortin (2013), *BSP-Why, a tool for deductive verification of BSP programs: machine-checked semantics and application to distributed state-space algorithms*
[25]: Herms (2013), *Certification of a Tool Chain for Deductive Program Verification*
[26]: Khan (2014), *Formal Specification and Verification of Computer Algebra Software*
[27]: Lehner et al. (2007), *Formal Translation of Bytecode into BoogiePL*
[28]: Vogels et al. (2009), *A Machine Checked Soundness Proof for an Intermediate Verification Language*
[29]: Backes et al. (2011), *Automatically Verifying Typing Constraints for a Data Processing Language*
[30]: Gössi (2016), *A Formal Semantics for Viper*

Vogels et al. [31] formalise and prove the soundness of a verifier that reduces an IVL program to a verification condition via a series of program-to-program transformations once and for all in Coq. Their supported programs form a simple Boogie subset and their program-to-program transformations are similar to those performed by the Boogie verifier. However, their Coq implementation does not consider various nontrivial aspects that are considered by practical verifier implementations such as Boogie's. For instance, they do not support loops or unstructured control flow and thus their transformations work naturally on an abstract syntax tree representation. In contrast, the Boogie verifier implementation switches early to a control-flow graph representation to deal with unstructured control flow (even if the input program uses structured control flow only), and thus differs significantly from the formalised Coq implementation. Moreover, Vogels et al. [31] do not support programs with background theory such as uninterpreted types and functions supported by Boogie, which simplifies the generation of verification conditions.

[31]: Vogels et al. (2010), *A machine-checked soundness proof for an efficient verification condition generator*

In addition to Vogels et al. [31], there are various works that formally prove once and for all the soundness of a translation from a program to VCs. This includes a simple VC generator [32], the generation of VCs from a language with implicit dynamic frames assertions [33], the generation of VCs for checking relational properties of programs with pointers [34], and the generation of VCs from a language similar to Why3 [25]. These proofs (on paper or in an ITP) do not directly connect to practical implementations.

[32]: Homeier et al. (1995), *A Mechanically Verified Verification Condition Generator*
[33]: Smans et al. (2012), *Implicit Dynamic Frames*
[34]: Blatter et al. (2022), *Certified Verification of Relational Properties*
[25]: Herms (2013), *Certification of a Tool Chain for Deductive Program Verification*

After reducing a program to a VC, some verifiers such as the Why3 verifier apply further logical transformations on the VC before handing the VC to a solver. We do not consider such logical transformations in this dissertation, since we apply our techniques to verifiers that do not apply such logical transformations for our considered language subsets. Garchery [35] develops a translation validation approach, which produces certificates on every run for some of Why3's logical translations. Cohen and Johnson-Freyd [36] prove some of Why3's logical transformations sound once and for all in Coq to demonstrate their Why3 semantics mechanisation. These works do not prove any part involved in reducing

[35]: Garchery (2021), *A Framework for Proof-carrying Logical Transformations*
[36]: Cohen et al. (2024), *A Formalization of Core Why3 in Coq*

the initial program to the VC before the logical transformations are applied.

Compilers

There has been a lot of work on formally validating compiler translations [37–41]. However, there are significant differences between compiler translations and program-to-program translations applied by translational programs verifiers. In particular, in contrast to compilers, translations applied by translational program verifiers incorporate reasoning steps, such as assumptions and proof obligations prescribed by a program logic. This encoding is achieved via components not present in executable languages such as `assume` commands, nondeterministic assignments, and axiomatisations (axiomatisations may, for instance, model parts of a state or model types such as arrays or sets). Moreover, for compiler translations, one must typically show that a single target execution corresponds to a single source execution, while for verifier translations the relationship is often different (*e.g.* a single source execution may be justified by the combination of a set of target executions).

[37]: Tristan et al. (2008), *Formal verification of translation validators: a case study on instruction scheduling optimizations*
[38]: Tristan et al. (2009), *Verified validation of lazy code motion*
[39]: Rizkallah et al. (2016), *A Framework for the Automatic Formal Verification of Refinement from Cogent to C*
[40]: Kang et al. (2018), *Crellvm: verified credible compilation for LLVM*
[41]: Gourdin et al. (2023), *Formally Verifying Optimizations with Block Simulations*

Formal results for other classes of automated program verifiers

While this dissertation focuses on formal results for translational program verifiers, there exist other classes of automated program verifiers, for which formal results have been shown. One popular verification approach is *symbolic execution*. Verifiers based on symbolic execution execute programs using symbolic values, and accumulate constraints on these values, which are then typically discharged using an SMT solver. Establishing the soundness of such verifiers requires different techniques than for translational program verifiers, since symbolic execution does not perform any program-to-program translations. However, symbolic execution can be used by IVL back-end verifiers, so translational program verifiers may ultimately depend on verifiers based on symbolic execution. In this dissertation, we consider only IVL back-end verifiers that ultimately produce VCs via a translation from a program to a VC that captures the entire verification condition of a program. Such a program-to-VC is translation is very different from symbolic execution, which queries the SMT solver at different points during a symbolic program execution, and performs certain checks directly without generating any logical conditions.

Verifiers based on symbolic execution are often implemented in mainstream programming languages. Some works formalise subsets of these verifiers (via a reimplementation) or represent the verifiers using an operational semantics, and then prove soundness on paper or in an ITP. Such results have been presented for Gillian [42], VeriFast [43] and Viper [44, 45]. Other works use translation validation approaches, which instrument existing implementations such that they generate certificates on every run. For instance, this includes the generation of Coq certificates for VeriFast [46], and the generation of Metamath [47] certificates for verifiers obtained via the K framework [48]. Finally, there are verifiers based on symbolic execution that are by default implemented and proved sound once and for all in an ITP. Examples for such verifiers based on

[42]: Lööw et al. (2024), *Compositional Symbolic Execution for Correctness and Incorrectness Reasoning (Extended Version)*

[43]: Vogels et al. (2015), *Featherweight VeriFast*

[44]: Zimmerman et al. (2024), *Sound Gradual Verification with Symbolic Execution*
[45]: Dardinier et al. (2025), *Formal Foundations for Translational Separation Logic Verifiers*

[46]: Wils et al. (2023), *Certifying C program correctness with respect to CH2O with VeriFast*

[47]: Megill et al. (2019), *Metamath: A Computer Language for Mathematical Proofs*

[48]: Lin et al. (2023), *Generating Proof Certificates for a Language-Agnostic Deductive Program Verifier*

symbolic execution include Katamaran [49] and VeriSmall [50]; both of these verifiers can be extracted from the ITP to an executable language.

[49]: Keuchel et al. (2022), *Verified symbolic execution with Kripke specification monads (and no meta-programming)*

[50]: Appel (2011), *VeriSmall: Verified Smallfoot Shape Analysis*

There are a variety of program verifiers that are built within an ITP, where automation is also achieved via symbolic execution, but where the symbolic execution is expressed via the metaprogramming and proof tactic facilities provided by ITPs. As a result, it is typically not possible to extract the verifier from the ITP. In these verifiers, a verification run corresponds to automatically finding a proof within the ITP. Thus, these verifiers are formally established to be sound by default. Examples of such verifiers include Bedrock [51], Diaframe [52], RefinedC [53], RefinedRust [54], and VST [55]. In some cases, such verifiers abstract over the ITP facilities via domain specific languages (DSLs) in which the automation is expressed (such as the Lithium DSL used by RefinedC and RefinedRust).

[51]: Chlipala (2011), *Mostly-automated verification of low-level programs in computational separation logic*

[52]: Mulder et al. (2022), *Diaframe: automated verification of fine-grained concurrent programs in Iris*

[53]: Sammler et al. (2021), *RefinedC: automating the foundational verification of C code with refined ownership types*

[54]: Gäher et al. (2024), *RefinedRust: A Type System for High-Assurance Verification of Rust Programs*

[55]: Cao et al. (2018), *VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs*

The proof-oriented programming language F* proposes another verification approach via a dependently-typed programming language. Verification in F* is automated by a type checker, which discharges logical conditions via an SMT solver. Strub *et al.* [56] develop a translation validation approach, which generates a certificate in Coq on every run, and apply their approach to a subset of F*. They use a self-certification approach: since the type checker is written in F*, it is sufficient to produce a single certificate to establish soundness for every run of the type checker (by running the type checker on itself). There are also languages built on top of F*, which can be used for automated verification, such as Steel [57], which supports separation logic specifications. Verification in Steel is proved sound against SteelCore [58] in F*. These works based on F* do not consider program-to-program translations applied by translational program verifiers.

[56]: Strub et al. (2012), *Self-certification: bootstrapping certified typecheckers in F* with Coq*

[57]: Fromherz et al. (2021), *Steel: proof-oriented programming in a dependently typed concurrent separation logic*

[58]: Swamy et al. (2020), *SteelCore: an extensible concurrent separation logic for effectful dependently typed programs*

1.3. Challenges

This dissertation goes beyond the state of the art by providing formal guarantees for existing and practical implementations of translational program verifiers, as opposed to formal guarantees for idealised implementations that are not used in practice. To do so, this dissertation must address the following high-level challenges:

Challenge 1: Dealing with existing implementations via certificates

It is practically infeasible to formally prove existing implementations used in practice sound once and for all, since they are large and are expressed in mainstream programming languages that lack formalisations and use diverse programming paradigms that are hard to reason about. As discussed earlier, this dissertation instead develops translation validation approaches that automatically produce a certificate on every run of the verifier implementation, which establishes soundness of the verifier run. As a result, this dissertation must identify suitable certificate representations. Moreover, it is practically infeasible to manually check the validity of generated certificates, since they are nontrivial and can be large. As a result, this dissertation must develop approaches that enable an ITP to *automatically* check the validity of the generated certificates, which is a challenge since ITPs themselves are primarily designed to be used in an interactive manner. Finally, this dissertation must instrument the existing implementation to obtain the necessary information to automatically generate certificates.

Challenge 2: Dealing with translations implemented in practice

Implementations of translational program verifiers that are used in practice apply optimisations typically not covered by idealised versions. In particular, verifier implementations used in practice translate certain features in different ways depending on the context in order to optimise encodings. The conditions under which the optimisations are sound may depend on nontrivial semantic conditions that the verifier checks by emitting explicit checks as part of program-to-program translations.

Moreover, implementations used in practice use translations that scale to a large subset of the to-be-verified source language. As a result, such implementations may use different program representations than idealised implementations, which results in new challenges that must be addressed. For instance, an implementation may switch from an abstract syntax tree representation to a control-flow graph representation in order to deal with unstructured control flow, and then performs all further translations on the control-flow graph. Idealised implementations often deal with structured control flow only and as a result only deal with abstract syntax tree representations.

Challenge 3: Bridging the large gap between source and target programs

The gap between source and target programs of translations applied by translational program verifiers is often large. This gap must be bridged when formally validating these translations. This large gap shows up for different reasons in different translations. For instance, the source and target languages themselves may be very different (*e.g.* in front-end translations), where the states, statements, and assertion languages may differ significantly. Another example is that a set of source executions may be justified by a set of target executions, where there is not a one-to-one correspondence between executions in the sets. For instance, in the translation eliminating cycles via loop invariants, a single source execution is justified by the combination of potentially infinitely many target executions. Yet another example for the large gap between the source and target programs is the program-to-VCs translation, since the program and VCs differ significantly.

In some cases, the state of the art also deals with this challenge. However, this dissertation must deal with this challenge *in addition* to the first two challenges, which are focused on providing guarantees for implementations used in practice. The interplay of this challenge with the other two leads to novel challenges not addressed by the state of the art. In particular, identifying suitable certificate representations and enabling the automatic checking of the validity of certificates is more challenging due to the large gap between source and target programs.

1.4. This Dissertation

This dissertation develops translation validation approaches for formally establishing the soundness of existing translational program verifiers used in practice. In particular, this dissertation applies these approaches to the widely-used Boogie [1] and Viper [7] verifier implementations. That is, we instrument both *existing* verifier implementations such that they automatically produce an Isabelle certificate on each run of the verifier for a core subset of Boogie and Viper programs.

[1]: Leino (2008), *This is Boogie 2*

[7]: Müller et al. (2016), *Viper: A Verification Infrastructure for Permission-Based Reasoning*

The Boogie verifier implementation translates a Boogie program to a VC by first applying a series of Boogie-to-Boogie transformations. Our produced certificate for a particular Boogie verifier run formally shows that if the VC is valid, then the input Boogie program conforms to its specification. This certificate combines separate certificates for the different transformations applied by Boogie; these certificates establish the soundness of the corresponding transformations separately. The Viper verifier implementation translates a Viper program to a Boogie program. Our produced certificate for a particular Viper verifier run formally shows that if the Boogie program conforms to its specification, then the Viper program conforms to its specification. In principle, one could combine the two certificates to obtain an end-to-end soundness result (if the VC is valid), but our certificate-producing support for the Boogie verifier must first be extended to deal with Boogie programs generated by the Viper verifier as we will elaborate on in Subsection 3.9.5.

This dissertation develops different translation validation approaches for different kinds of translations. In particular, the approach developed for the Viper-to-Boogie translation differs from the approaches developed for the Boogie-to-Boogie transformations and the final generation of the VC from a Boogie program. Each approach is designed in a general way such that the approach can be applied to similar translations. For instance, the approach developed for the Viper-to-Boogie translation is designed to also work for other front-end translations. Overall, this dissertation applies these approaches to a wide variety of translations including (1) a front-end translation (the Viper-to-Boogie translation), (2) two nontrivial translations within the same language (cycle elimination and assignment elimination applied by the Boogie verifier), and (3) the generation of VCs (the final generation of the VC applied by the Boogie verifier).

This dissertation's validation approaches are designed to enable the automation of two core aspects: (1) the automatic *generation* of certificates, and (2) the automatic *checking* of generated certificates. Both of these aspects are fully automatic in our certificate-producing support for the Boogie and Viper verifiers. That is, there is no user interaction required in the generation and checking of certificates. Our instrumentations of the verifiers enable the automatic generation of certificates, which contain sufficient information for Isabelle to automatically check them. The generated certificates contain fine-grained Isabelle *tactics*, which provide instructions for *how* to prove formal statements. Many of the tactics in our certificates are straightforward for Isabelle to successfully execute as desired (such as a tactic that specifies the application of a lemma). Some of the tactics in our certificates are more involved and lead to Isabelle itself performing some sort proof search. In theory, in this latter case, there is no guarantee that Isabelle successfully executes these tactics. However, in practice, as our evaluation shows, Isabelle succeeds almost always,[1] and thus, successfully checks our generated certificates automatically.

1: In the case of Boogie, there are rare cases where Isabelle is not able to successfully execute some tactics as desired; it is clear how to extend our work to fix these cases.

In summary, we make the following high-level contributions in this dissertation:

- We develop general translation validation approaches for a large variety of translations applied by translational program verifiers.
- We apply these general approaches to existing verifier implementations that are used in practice. This application results in certificate-producing instrumentations of the Boogie and Viper verifier implementations. The generation and subsequent checking of these certificates is fully automatic.
- To enable trustworthy certificates, we mechanise subsets of the Boogie and Viper languages in Isabelle. For both Boogie and Viper, these are the first mechanisations of the chosen subsets.

Impact on Boogie and Viper ecosystems

In addition to the mentioned contributions, this dissertation's work has led to insights that have had a positive impact on the Boogie and Viper ecosystems. We discuss some of this impact in more detail in Section 3.8 and Chapter 4. At a high level, our work on Boogie enabled discussions on nontrivial Boogie features, where alternative encodings were being explored by Boogie developers. Moreover, our work also enabled reasoning formally about translations into Boogie, which led to novel insights into the existing Dafny-to-Boogie and Viper-to-Boogie translations. For instance, we improved the existing Viper-to-Boogie translation along multiple dimensions: (1) fixing a previously undiscovered soundness issue, and (2) improving the code in general (*e.g.* leading to better error reporting and to a cleaner implementation).

Our work on Viper also helped gain important insights into the semantics of Viper in general. While we point out insights specific to this dissertation here, projects that were led by Thibault Dardinier also generated significant insights into the semantics that are not presented in this dissertation. In general, the overall work on providing formal foundations for the Viper ecosystem was a collaborative effort. This collaborative effort of understanding the Viper semantics as a whole (*e.g.* beyond the Viper subset considered in this dissertation) was crucial to understand how to define the semantics in this dissertation such that the work presented here is extensible to larger Viper subsets. In particular, this effort revealed that one had to take certain design decisions in the semantics, which were not known before, and also clarified the semantics of various nontrivial Viper features and their interactions. Finally, the work presented in this dissertation is important to make sure that the Viper verifier implementation itself respects the semantics we had in mind.

Outline

Chapter 2 presents our formal validation approach for the existing Boogie verifier. Chapter 3 presents our formal validation approach for front-end translations and applies this approach to the existing Viper-to-Boogie translation. Finally, Chapter 4 concludes.

The chapters present additional information, which is not required to follow the main text, but which are nevertheless relevant (such as alternative approaches and additional context information), in blue boxes such as the following:

Title for additional information

Text describing additional information

1.5. Publications and Collaborations

The main results of this dissertation were presented in two separate publications.

The main results of Chapter 2 were presented in:

Gaurav Parthasarathy, Peter Müller, Alexander J. Summers.
Formally Validating a Practical Verification Condition Generator
In Computer-Aided Verification (CAV) 2021 [59]

The main results of Chapter 3 were presented in:

Gaurav Parthasarathy, Thibault Dardinier, Benjamin Bonneau, Peter Müller, Alexander J. Summers.
Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language
In Proceedings of the ACM Programming Languages (PLDI) 2024 [60]

The work in this dissertation, including the writing and technical work in the publications, was led by me.[2] In addition to my supervisors, I collaborated with various people who contributed to parts of this dissertation. Aleksandar Hubanov and Lukas Himmelreich helped with the formal validation of Boogie as part of Bachelor's theses supervised by me [61, 62] (this work happened after the corresponding publication [59]). Aleksandar implemented the formal validation for the AST-to-CFG transformation briefly discussed in Section 2.9, and Lukas implemented the formal validation for the CFG optimisations discussed in Section 2.7. I did all of the remaining formal validation work on Boogie. Alain Delaët–Tixeuil's internship project cosupervised by me made it clearer to me that formally validating the Boogie implementation is a worthwhile research project. I did most of the work for the Viper-to-Boogie translation. However, two people had an important impact on this work. Thibault Dardinier helped with numerous discussions on formal models of Viper in general and on the formal validation of the translation. Thibault also helped with the evaluation of the work. As part of an internship project cosupervised by me, Benjamin Bonneau explored the formal validation of the Dafny-to-Boogie translation [63]. This work helped identify certain aspects that needed to be generalised for a practical validation approach and helped choose the direction for the Viper-to-Boogie work. Moreover, I adapted the state relation instantiation and the Boogie abstract value instantiation from Benjamin's work to the Viper-to-Boogie setting (see Subsection 3.5.1 and Subsection 3.3.7).

2: Here, I use first person singular to distinguish my contributions.

[61]: Hubanov (2022), *Formally Validating the AST-to-CFG Phase of the Boogie Program Verifier*
[62]: Himmelreich (2023), *Formally Validating the CFG Optimization Phase of the Boogie Program Verifier*

[63]: Bonneau (2021), *A formal foundation for the Dafny verifier*

Contributions beyond this dissertation

In the time working on this dissertation, I contributed to the following publications, which were led by other researchers:

Thibault Dardinier, Michael Sammler, Gaurav Parthasarathy, Alexander J. Summers, and Peter Müller
Formal Foundations for Translational Separation Logic Verifiers
In Proceedings of the ACM Programming Languages (POPL) 2025 [45]

Thibault Dardinier, Gaurav Parthasarathy, Peter Müller
Verification-Preserving Inlining in Automatic Separation Logic Verifiers
In Proceedings of the ACM Programming Languages (OOPSLA) 2023 [64]

Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, Alexander J. Summers
Sound Automation of Magic Wands
In Computer-Aided Verification (CAV) 2022 [65]

Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, Bart Jacobs
The Future is Ours: Prophecy Variables in Separation Logic
In Proceedings of the ACM Programming Languages (POPL) 2020 [66]

Formally Validating a Verification Condition Generator 2.

2.1. Introduction

The Boogie verifier is a translational verifier: it performs a sequence of substantial Boogie-to-Boogie transformations followed by the final generation of a *verification condition* (VC). The verification condition is then discharged by a SMT solver. The main purpose of the applied transformations is to adjust the program in order to optimise or simplify later transformations, and to generate a space-efficient VC (based on the technique described by Barnett and Leino [23]). Since the Boogie verifier contains all transformations from the input to the VC, the verifier itself is a *verification condition generator*.

[23]: Barnett et al. (2005), *Weakest-precondition of unstructured programs*

In this chapter, we develop techniques to establish the soundness of the *existing* Boogie verifier implementation. The Boogie verifier is sound if the input Boogie program is *correct* whenever the SMT solver reports that the generated VC is valid. Existing work that establishes the soundness of translational program verifiers is based on idealised implementations that are formalised on paper or in an interactive theorem prover. As discussed in Chapter 1, there is a large gap between these implementations and existing verifier implementations used in practice which can and do exhibit soundness bugs. In this chapter, we bridge this gap by developing an approach to formally validate runs of the existing Boogie verifier implementation.

Proving the existing Boogie verifier implementation once and for all is practically infeasible, since it consists of over 30K lines of code, and is written in C#, a language that lacks a formalisation. Instead, we use a formal translation validation approach: we instrument the *existing* Boogie verifier implementation such that on every run of the verifier a certificate is generated. The generated certificate formally establishes soundness of the corresponding verifier run under the assumption that the generated VC is valid. Certification of the validity-checking of the VC is an orthogonal concern; our results can be combined with work in that area (see [8–10]) to obtain end-to-end guarantees. Our generated certificates are expressed in the Isabelle theorem prover [12] and connect directly to an operational semantics of input Boogie programs. Thus, our certificates provide formal and trustworthy guarantees. Moreover, we include sufficient information in the certificates such that Isabelle is able to automatically check them. Ensuring the automatic checking of certificates is crucial, since the size and complexity of certificates make manually checking them practically infeasible.

[8]: Böhme et al. (2010), *Fast LCF-Style Proof Reconstruction for Z3*
[9]: Ekici et al. (2017), *SMTCoq: A Plug-In for Integrating SMT Solvers into Coq*
[10]: Fleury et al. (2019), *Reconstructing veriT Proofs in Isabelle/HOL*

[12]: Nipkow et al. (2002), *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*

The key challenges in certifying runs of the Boogie verifier are to certify each of the transformations applied by the verifier, including the final generation of the VC. In particular, we present novel techniques for making the following three key transformations (and many smaller ones) of Boogie's tool chain certifying (the final certificate uses all of these techniques to establish an end-to-end soundness result incorporating all transformations):

1. The elimination of loops (more precisely, cycles in the control-flow graph) by reducing the correctness of loops to checking loop invariants.
2. The elimination of assignments by (static-single-assignment-style) introduction of fresh variables and suitable **assume** commands.
3. The final generation of the VC, which includes the erasure and logical encoding of Boogie's polymorphic type system [67].

[67]: Leino et al. (2010), *A Polymorphic Intermediate Verification Language: Design and Logical Encoding*

The formal validation of the two program-to-program transformations is related to existing work on compiler verification [68] and validation [37, 69]. However, both key Boogie-to-Boogie transformations and the certified soundness property of the transformations that we tackle here are fundamentally different from those in compilers. Compilers typically require that each execution of the target program corresponds to an execution of the source program. As a result, compiler translations typically do not introduce nondeterminism. In contrast, cycle elimination and assignment elimination introduce nondeterminism via nondeterministic assignments and **assume** commands. In the case of cycle elimination, a *set* of target executions together justify a single source execution. Prior work on validating such verifier transformations has been limited in the supported language and extent of the formal guarantee; we discuss comparisons in detail in Section 2.11.

[68]: Leroy (2006), *Formal certification of a compiler back-end or: programming a compiler with a proof assistant*

[37]: Tristan et al. (2008), *Formal verification of translation validators: a case study on instruction scheduling optimizations*

[69]: Barthe et al. (2014), *Formal Verification of an SSA-Based Middle-End for CompCert*

Contributions

This chapter makes the following technical contributions:

1. The first formal semantics for a significant subset of Boogie (including axioms, polymorphism, type constructors, and type quantification) that is mechanised in Isabelle.
2. A validation technique for two core program-to-program transformations occurring in verifiers (cycle elimination and assignment elimination), along with validation techniques for smaller transformations such as the transformation from an abstract syntax tree to a control-flow graph representation and the coalescing of basic blocks in a control-flow graph. All of these validation techniques follow the same general principle.
3. A validation technique for the final generation of the VC, handling polymorphism erasure and Boogie's type system encoding [67], for which no prior formal proof exists. This validation technique follows the same general principle as the techniques developed for the program-to-program transformations.
4. An instrumentation of the existing Boogie implementation that produces automatically checkable certificates for the subset of Boogie that we formalise.

Making the Boogie verifier certifying is an important result, reducing the trusted code base for a wide variety of translational program verifiers implemented via translations into Boogie, such as Corral [70], Dafny [3], SMACK [16], SYMDIFF [71], VCC [72], and Viper [7]. Moreover, the technical approaches we present here could be adapted to other translational verifiers that use similar translations.

[70]: Lal et al. (2012), *A Solver for Reachability Modulo Theories*

[3]: Leino (2010), *Dafny: An Automatic Program Verifier for Functional Correctness*

[16]: Carter et al. (2016), *SMACK software verification toolchain*

[71]: Lahiri et al. (2012), *SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs*

[72]: Cohen et al. (2009), *VCC: A Practical System for Verifying Concurrent C*

[7]: Müller et al. (2016), *Viper: A Verification Infrastructure for Permission-Based Reasoning*

Outline

Section 2.2 discusses the translations applied by the Boogie verifier and how our validation approach is structured at a high level. Section 2.3 introduces a formal semantics for Boogie control-flow graphs. Section 2.4, Section 2.5 and Section 2.6 present our validation of cycle elimination, assignment elimination, and the final generation of the VC, respectively. Section 2.7 presents our validation of control-flow graph optimisations. Section 2.8 introduces a formal semantics for Boogie abstract syntax trees and Section 2.9 briefly discusses our validation of the translation from an abstract syntax tree to a control-flow graph. Section 2.10 evaluates our certificate-producing version of Boogie. Section 2.11 discusses related work and Section 2.12 discusses future directions.

Access to tool and Isabelle formalisation

Our certificate-producing version of Boogie is available online:

- Repository:
  `https://github.com/viperproject/boogie-proofgen`
- Branch for dissertation: dissertation-gaurav
- Commit hash at time of dissertation submission:
  b7e20dc43633ff02cef180de544cef15a1a3be43

The main code for generating certificates is in the *Source/ProofGeneration* folder, which contains exclusively new code added by us. The existing Boogie verifier itself is spread across the remaining subfolders of *Source*. For instance, *Source/BoogieDriver* contains the entry point of the verifier, *Source/Core* contains the abstract syntax tree representation for Boogie programs and code for some of the earlier transformations in Boogie's pipeline, and *Source/VCGeneration* contains code for the later transformations including assignment elimination and final VC generation. We have instrumented parts of these folders containing the existing Boogie verifier in order to obtain sufficient information to generate certificates. The lines of added code for the instrumentation is significantly smaller than the code that actually generates certificates (the latter is in *Source/ProofGeneration*). A large part of the code added as part of the instrumentation invokes methods in *Source/ProofGeneration/ProofGenerationLayer.cs*, which serves as an interface between the existing Boogie verifier and the code generating certificates.

Our formal Boogie semantics and the metatheory used by the certificates is available online [73]:

- Repository:
  `https://github.com/viperproject/foundational-boogie`
- Branch for dissertation: dissertation-gaurav
- Commit hash at time of dissertation submission:
  90411340ac568c7870e85dd9ec627b84f01e79a3

2.2. High-Level Validation Approach

A Boogie program consists of global declarations and procedures. Global declarations model the background theory (for example, axiomatising types not supported directly by Boogie), and define global variables and constants. Each procedure has a specification (a pre- and postcondition) and a procedure body. Boogie verifies each procedure modularly, desugaring procedure calls according to their specifications. Verification of each procedure is implemented via a sequence of transformations: procedure-to-procedure transformations and a final computation of a verification condition (VC) to be checked by an SMT solver. Our goal is to formally certify (per run of Boogie) that the conjunction of the validity of the VCs for each procedure implies the correctness of the original program.

To keep the complexity of certificates manageable, our technical approach is *modular* in three dimensions. First, we generate certificates for each *procedure* in the Boogie program separately. Second, for a given procedure, we generate certificates for different subsequences of *transformations* separately. Third, for a sequence of transformations, we generate certificates for *smaller parts of the corresponding source procedure body and target procedure body (or target VC)* separately (*e.g.* blocks in a control-flow graph); the high-level idea for this decomposition is the same for all sequences of transformations that we consider. This modularity makes the full automation of checking certificates in Isabelle practical. In the following, we give a high-level overview of this modular structure. At the end (see Subsection 2.2.4), we show a small snippet of a concrete Isabelle certificate automatically generated by our tool. The formal semantics of Boogie programs and the details of the certificate generation are presented in subsequent sections.

2.2.1. Procedure Decomposition

Boogie has no notion of a main procedure or an overall program execution. A Boogie program is correct if each of its procedures is individually correct w.r.t. the global declarations in the program (which includes user-provided axioms and uninterpreted functions). A procedure is correct if its body has no failing executions, as we make precise in Section 2.3. Boogie computes a separate VC for each procedure, and we correspondingly validate the verification of each procedure separately. That is, we generate a certificate showing that the validity of the VC generated for a procedure implies that the procedure in the original program is correct.

2.2.2. Transformation Decomposition

Boogie parses an input program into an abstract syntax tree (AST) representation and then applies the sequence of transformations shown by the solid (black) edges in Figure 2.1 to each procedure. (If input programs have certain features outside of our supported Boogie subset, then Boogie applies more transformations, which we do not show in Figure 2.1.) The first transformation constructs a control-flow graph (CFG) from the

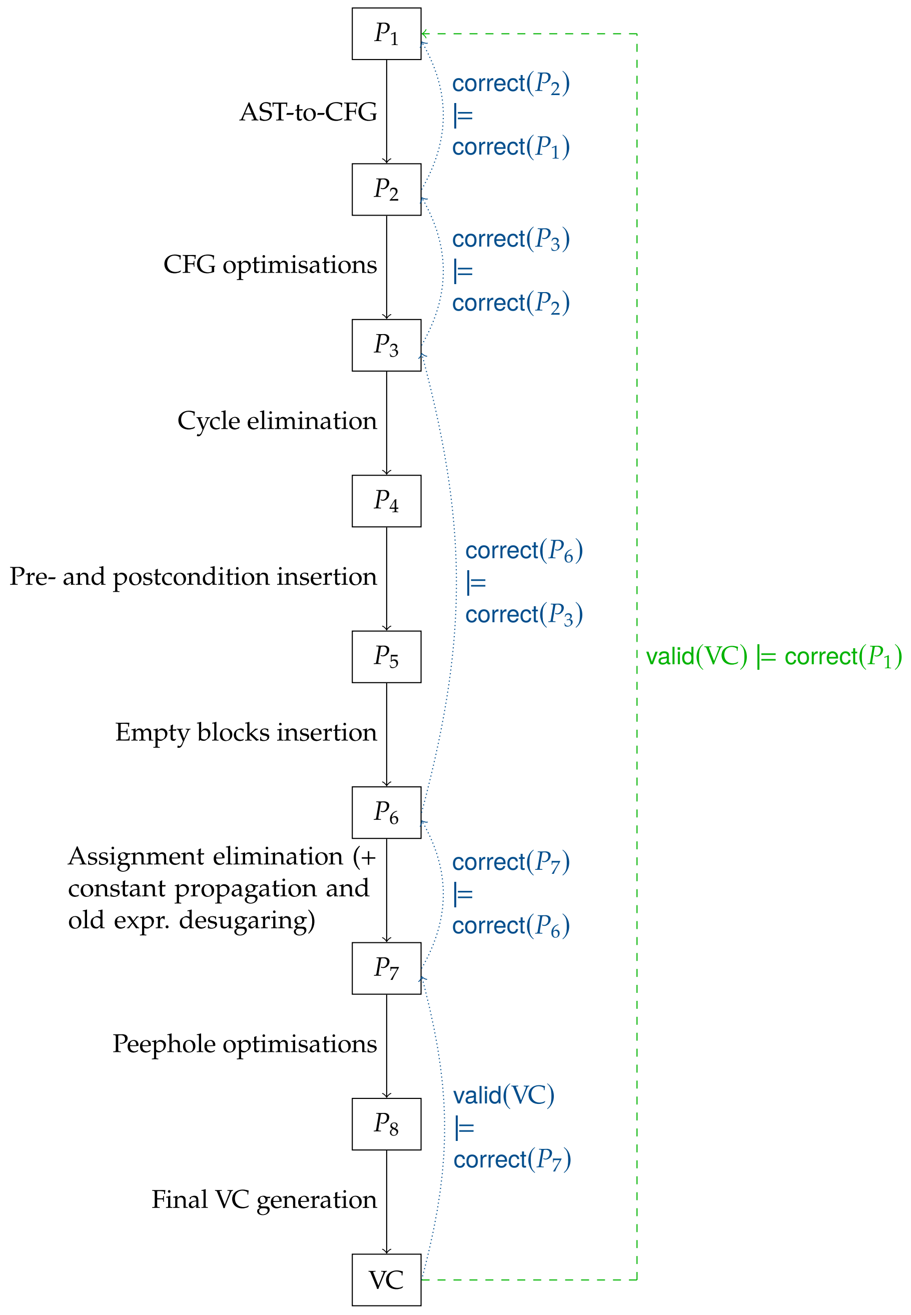

P_1
AST-to-CFG
P_2
CFG optimisations
P_3
Cycle elimination
P_4
Pre- and postcondition insertion
P_5
Empty blocks insertion
P_6
Assignment elimination (+ constant propagation and old expr. desugaring)
P_7
Peephole optimisations
P_8
Final VC generation
VC
correct(P_2) \|= correct(P_1)
correct(P_3) \|= correct(P_2)
correct(P_6) \|= correct(P_3)
correct(P_7) \|= correct(P_6)
valid(VC) \|= correct(P_7)
valid(VC) \|= correct(P_1)

Figure 2.1: Transformations applied by Boogie to a procedure and the certificates generated by our certificate-producing version of Boogie. The solid edges show Boogie's transformations on a procedure body. The node P_1 represents the Boogie abstract syntax tree of a procedure in the input program. The other nodes P_i ($2 \leq i \leq 8$) represent Boogie control-flow graphs, and the node VC represents the verification condition. Our final certificate (dashed edge in green) is constructed by formally linking the transformation certificates represented by the dotted edges in blue. Each transformation certificate formally validates one or more transformations.

AST. The source procedures of all the subsequent transformations are represented as CFGs.

We generate independent soundness certificates for different subsequences of transformations, essentially showing that the correctness of the target procedure (or validity in case of the VC) *after* the application of the transformations implies the correctness of the source procedure *before* the application of the transformations. In Figure 2.1, these certificates are shown via the dotted (blue) edges. For example, we certify that the correctness of the procedure P_6 (obtained after the insertion of empty blocks) implies the correctness of the procedure P_3 (obtained before applying the elimination of cycles). This certificate justifies a subsequence of three transformations. Finally, we glue these certificates together to guarantee the end-to-end property for the entire pipeline, namely that the validity of the VC implies that the input procedure is correct, as shown by the dashed (green) edge in Figure 2.1.[1] For our certificates, we import the related source and target procedures (or target VC) of each subsequence of transformations from Boogie into Isabelle; we do not reimplement any of Boogie's transformations inside Isabelle.

1: We currently trust the parser. That is, we do not prove a formal connection between the AST and the textual representation of the program. The trustworthiness of the Boogie parser becomes irrelevant for front-end translations into Boogie if one proves a connection between a front-end program and Boogie's AST program (as we will do in Chapter 3).

Our modular approach lets us treat transformations that have fundamentally different challenges separately, which makes the certification strategies simpler. Moreover, certificates for a subsequence of transformations are robust to *changes* to other transformations. Both of these arguments (simpler certification strategies and robustness) become stronger if one generates a separate certificate for each *single* transformation instead of considering larger subsequences. However, considering single transformations comes at a cost: one must represent every intermediate program in the pipeline explicitly in the certificates, which leads to larger certificates and thus slower certificate checking. By considering larger subsequences, we make a trade-off: we choose slightly more complex certificates and we are less robust to changes, but instead get smaller certificates. For example, in our certificate relating P_6 with P_3 in Figure 2.1, the intermediate procedures P_4 and P_5 are never made explicit in the certificate (and thus not imported into Isabelle).

Boogie's transformations

We now give an overview of the transformations shown in Figure 2.1 and motivate our choice for the different transformation subsequences that we consider in separate certificates.

The three most substantial and technically-challenging transformations are *cycle elimination*, *assignment elimination*, and final *VC generation*. Since each of these transformations has very different challenges, we make sure that they are handled in different certificates for different transformation subsequences.

Cycle elimination translates a CFG to an acyclic CFG (see Section 2.4). This transformation substantially alters the possible executions through the target CFG compared to the source CFG by *cutting loops* using annotated loop invariants to overapproximate the executions in the source CFG. The transformation cuts the loops by eliminating *back edges*, that is, edges from the end of a loop to the loop's entry. To identify loops and their corresponding back edges in a control-flow graph, Boogie computes *dominators* for blocks in the CFG, which is a nontrivial computation.[2]

2: A block b *dominates* a block b' if every path from the entry block of the procedure to b' visits b.

Our certificate for cycle elimination includes a transformation that inserts pre- and postconditions, and a transformation that inserts empty blocks. Including these these smaller transformations does not add much complexity to the generated certificate. Boogie inserts pre- and postconditions explicitly into the CFG via `assume` and `assert` commands (potentially adding new blocks to the CFG). Boogie inserts empty blocks to ensure that there are no *join blocks* (*i.e.* blocks with multiple predecessors) that have a predecessor with multiple successors. This is required to ensure that the elimination of assignments is sound in general, as we will discuss in Subsection 2.5.6. We discuss these transformations and the corresponding certification strategy in Section 2.4.

Assignment elimination removes imperative updates by transforming the code into static single assignment (SSA) form and then replacing assignments with *constraints* on variable versions (see Section 2.5). This elimination yields a *passified* CFG, that is, a CFG without any updates to variables.[3] Assignment elimination introduces extra nondeterminism and `assume` commands (which, if implemented incorrectly could make verification unsound by masking errors in the program), and changes the set of local variables in the procedure substantially, which complicates the state relationship between executions before and after the transformation.

3: As a result, assignment elimination is sometimes called *passification* in the literature.

Assignment elimination is implemented simultaneously with two other changes to the procedure: (1) constant propagation, and (2) old expression desugaring (expressions where the global variables refer to their values as they were at the beginning of the corresponding procedure). As a result, to avoid any further complexity, we generate a certificate just for assignment elimination together with constant propagation and old expression desugaring without incorporating any other transformations. We discuss assignment elimination and our corresponding certification strategy in Section 2.5.

The final generation of the VC translates the acyclic, passified CFG to a verification condition that, in addition to capturing the weakest precondition of the source procedure, encodes away Boogie's polymorphic type system [67]. Our certificate for the final generation of the VC includes the preceding peephole optimisations, which do not add much complexity. These peephole optimisations just prune unreachable blocks and remove empty blocks. We discuss the details of the final generation of the VC and our corresponding certification strategy in Section 2.6.

[67]: Leino et al. (2010), *A Polymorphic Intermediate Verification Language: Design and Logical Encoding*

In addition to these three certificates for the three discussed subsequences, we generate separate certificates for the AST-to-CFG transformation and the CFG optimisations. The CFG optimisations coalesce blocks and prune unreachable blocks. The coalescing of blocks requires some special care in the certificates and is one reason why we deal with the CFG optimisations in a separate certificate. Another reason is that we want the certificate generation for the AST-to-CFG transformation to be unaffected even if more complex CFG optimisations were added.

The CFG optimisations contain two separate transformations that we combine in a single certificate. (Figure 2.1 shows these two separate transformations as a single transformation for the sake of presentation). The first transformation prunes certain blocks that cannot be reached by any execution and the second transformation coalesces certain blocks. Additionally, the CFG optimisations contain a third transformation that eliminates dead variables. Our certificates do not support this third transformation yet. If there are no dead variables, then we are still able to generate the certificate as shown in Figure 2.1. If there are dead variables that Boogie eliminates, then we obtain a weaker result: we show the soundness of the AST-to-CFG transformation and that the validity of the VC implies the correctness of the CFG after the AST-to-CFG transformation but where the dead variables are eliminated. We elaborate on this in Section 2.7. It is future work to connect these two results by validating the dead variable elimination to obtain the end-to-end theorem even if dead variables are eliminated; doing so should be straightforward.

The implementation of Boogie contains several optional transformations and transformations for Boogie features not yet supported by our generated certificates. For instance, we do not support Boogie maps, for which Boogie has separate transformations. As another example, we do not support the validation for Boogie's support for loop invariant inference via abstract interpretation, which can be enabled via a command-line option. Note that Boogie by default does not check that inferred loop invariants are correct loop invariants and instead considers them to be *free loop invariants* that are just assumed. Thus, to support Boogie's loop invariant inference, one would have to show that the inference infers only correct loop invariants. Our subsets for CFGs and ASTs will be made clear in Section 2.3 and Section 2.8, respectively. We support only the default command-line options except for the type system encoding that we discuss in Section 2.6.

2.2.3. Procedure Body Decomposition

For a subsequence of transformations, our goal is to automatically generate certificates which formally prove that the correctness of the target procedure (or validity of the VC) implies the correctness of the source procedure. This is equivalent to proving that if the target procedure has no failing executions (or the VC is valid), then neither does the source procedure. When tackling the proof of this goal for a subsequence of transformations, we further break down the proof into smaller problems, each focusing on a part of the source and target procedure body.

We must consider three kinds of source-target pairs: (1) an AST source and a CFG target, (2) a CFG source and CFG target, and (3) a CFG source and a VC target. We first discuss our approach for the case when both the source and target are represented as CFGs and then show the generalisation to the other two cases.

Relating a CFG source with a CFG target

For the most part, every block in the source CFG has a unique corresponding block in the target CFG, and vice versa. However, there are exceptions, for example, when (1) blocks are coalesced, (2) new blocks are introduced, or (3) blocks are removed. We ignore these exceptions in this section. The presented approach can be generalised to the exceptions and we will discuss how we handle them in later sections.

For every block B_s in the source CFG and its corresponding block B_t in the target CFG, we prove two results that each relate the two blocks:

1. *Local block lemmas:* We relate executions that go through B_s with executions through B_t. This result is proved in isolation from any of the other blocks and thus all local block lemmas can be proved in parallel.
2. *Global block theorems:* We relate executions starting from B_s and extending to the rest of the source CFG with executions starting from B_t and extending to the rest of the target CFG.

This decomposition separates command-level reasoning (local block lemmas) from CFG-level reasoning (global block theorems). It enables concise lemmas and proofs in Isabelle and makes each comprehensible to a human. This separation also makes, for instance, the CFG-level reasoning robust to changes that affect only the command-level reasoning.

The details of local block lemmas and global block theorems differ for the different transformation subsequences. However, in all cases, the essence of the lemmas is the same. Local block lemmas imply two key properties if there are no failing executions through the target block from a state σ_t: (1) there are no failing executions through the corresponding source block from states related to σ_t, and (2) given a *successful* execution through the source block starting from a state related to σ_t and ending in a state σ'_s, there must exist a corresponding successful execution through the target block starting from σ_t and ending in a state related to σ'_s. The first property is necessary to ensure that the procedure is correct and we use the second property to compose local block lemmas in order to obtain the global block theorems.

Global block theorems generalise the first property ensured by local block lemmas to entire CFGs: if there are no failing executions in the target CFG starting from the target block from state σ_t (possibly extending to the rest of the target CFG), then there are no failing executions in the source CFG starting from any state related to σ_t. The soundness of the transformation (*i.e.* the absence of failing executions in the target CFG implies the absence of failing executions in the source CFG) follows essentially from (1) the global block theorem for the entry block, and (2) a proof showing that there is a related initial target state for any initial source state.

Note that the described properties implied by the local block lemmas and global block theorems essentially capture a *forward simulation* [74] between the source and target procedures. However, for cycle elimination and assignment elimination, our local block lemmas and global block theorems use simulation techniques that go beyond traditional forward simulations which construct a single target execution for a given source execution. For cycle elimination, a looping source execution must be justified by *multiple* non-looping target executions. For assignment elimination, our approach tracks *multiple* target executions for a single source execution in order to split the proof modularly. We will discuss both of these in later sections.

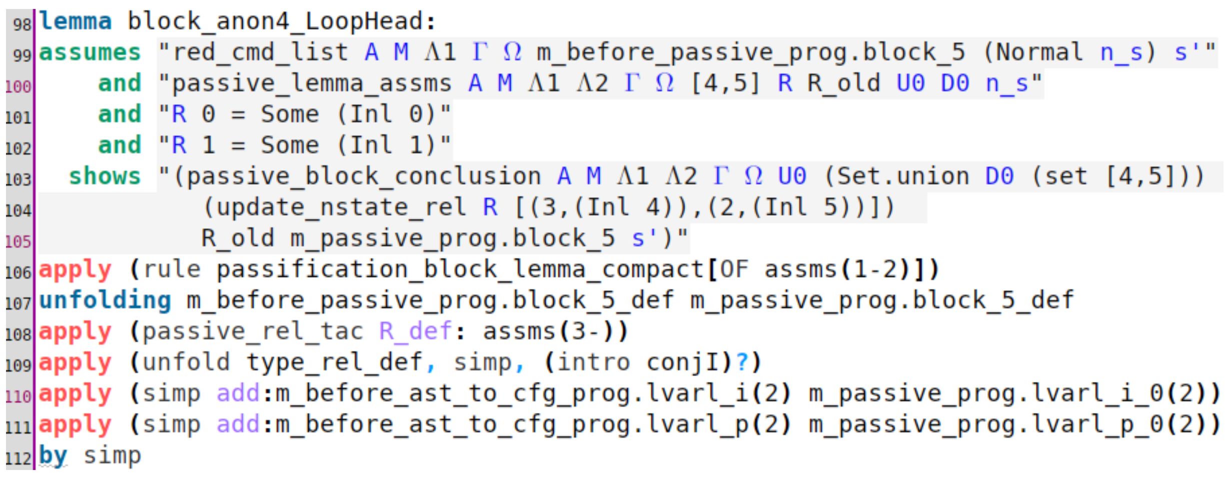
[74]: Lynch et al. (1995), *Forward and Backward Simulations: I. Untimed Systems*

For the different transformation subsequences, the details of the local block lemmas (and thus global block theorems) differ for various reasons. In some cases one requires extra conditions on the input states or the output states. For example, for the proof involving cycle elimination, we may need to assume in certain cases that a loop invariant holds in the input state or we may have to prove that the loop invariant holds in the output state. Another difference in the local block lemmas is regarding the existence of target executions. Above we presented the concept of a local block lemma as requiring the existence of a *single* target execution for every successful source execution. However, for assignment elimination, the local block lemmas state a stronger result that requires proving the existence of potentially infinitely many target executions. We will discuss the reasons in Section 2.5.

Proving a global block theorem

To prove a global block theorem for a source block B_s and corresponding target block B_t, we use the local block lemma relating B_s and B_t, and the global block theorems of the successor blocks. This proof strategy thus induces a set of dependencies between global block theorems. It is crucial that these dependencies do not form cycles, otherwise one cannot formally establish the global block theorem of the entry block, which is the main global block theorem we want to finally prove. Isabelle (and ITPs in general) guarantee that such circular reasoning does not occur, because one can use a Isabelle lemma only if it already has been proved. Thus, when generating certificates, we are forced by Isabelle to prove the global block theorems in an order where before the theorem for a block pair is proved, all the global block theorems for the corresponding successors are proved. If the source and target CFGs are acyclic (which is the case from cycle elimination onwards), one can do so using a reverse-topological order (*i.e.* starting from the exit blocks, which have no successors, and then moving backwards through the CFG). If the source or the target CFG has cycles, then the order is more involved. In this case, the intuition is to move backwards through the CFG and handle cycles via induction proofs. One must distinguish whether the cycles in the source and target CFG are in sync (*e.g.* for the CFG optimisations) or not (for the subsequence including cycle elimination). We will discuss the details in the respective sections.

The high-level proof idea for a global block theorem is the following for the case where a single target execution simulates a given source execution (the other cases are similar). To show the theorem in this case, we assume the existence of a failing source execution (starting from a source block B_s) and prove that this implies a failing target execution (starting from the corresponding target block B_t). If the source execution fails during the execution of B_s, then we can use the corresponding local block lemma to directly find a failing execution through B_t, which concludes the proof. Otherwise, the source execution successfully goes through B_s reaching some state σ'_s and then will fail starting from a successor B'_s of B_s. In this case, we can use the local block lemma to obtain a successful target execution e_t through B_t to reach a state related to σ'_s. Then, we can use the global block theorem of the successors (for B'_s and the corresponding target block B'_t) and the failing execution from



```
98  lemma block_anon4_LoopHead:
99  assumes "red_cmd_list A M Λ1 Γ Ω m_before_passive_prog.block_5 (Normal n_s) s'"
100     and "passive_lemma_assms A M Λ1 Λ2 Γ Ω [4,5] R R_old U0 D0 n_s"
101     and "R 0 = Some (Inl 0)"
102     and "R 1 = Some (Inl 1)"
103   shows "(passive_block_conclusion A M Λ1 Λ2 Γ Ω U0 (Set.union D0 (set [4,5]))
104            (update_nstate_rel R [(3,(Inl 4)),(2,(Inl 5))])
105            R_old m_passive_prog.block_5 s')"
106 apply (rule passification_block_lemma_compact[OF assms(1-2)])
107 unfolding m_before_passive_prog.block_5_def m_passive_prog.block_5_def
108 apply (passive_rel_tac R_def: assms(3-))
109 apply (unfold type_rel_def, simp, (intro conjI)?)
110 apply (simp add:m_before_ast_to_cfg_prog.lvarl_i(2) m_passive_prog.lvarl_i_0(2))
111 apply (simp add:m_before_ast_to_cfg_prog.lvarl_p(2) m_passive_prog.lvarl_p_0(2))
112 by simp
```

Figure 2.2: A snippet of an automatically generated Isabelle certificate for a Boogie program. The Isabelle lemma in this snippet expresses a local block lemma relating a concrete source and target block as part of assignment elimination. Lines 98-105 formally state the local block lemma and lines 106-112 form the proof of the lemma. All applied tactics in the proof are built-in Isabelle tactics (such as the `rule` and `simp` tactics) except for the `passive_rel_tac` tactic, which is a general custom tactic that we defined and that we use as part of automatically generated assignment elimination certificates. The `passive_rel_tac` tactic itself applies built-in Isabelle tactics.

B_s' to prove the existence of a failing target execution e_t' starting from the successor B_t'. The composition of e_t and e_t' provides a failing target execution from B_t, which concludes the proof.

Relating an AST or VC with a CFG

We generalise our approach for relating source and target CFGs to relating a source AST with a target CFG and a source CFG with a target VC. In the former case, the local block lemmas relate a sequence of basic commands in the AST with a block in the CFG. The global block theorems relate a program point in the AST with a block in the CFG. For the latter case, we exploit the fact that Boogie generates a separate verification condition for each block [23]. In particular, we design our local block lemmas and global block theorems such that they relate CFG blocks with their corresponding verification condition. We will discuss our certification of the final generation of the VC in Section 2.6.

[23]: Barnett et al. (2005), *Weakest-precondition of unstructured programs*

2.2.4. A Snippet of a Concrete Certificate in Isabelle

To make clearer how our generated Isabelle certificates look at a high level, consider Figure 2.2, which shows a snippet of an Isabelle certificate automatically generated by our tool for some input Boogie program. In particular, this snippet shows a local block lemma (along with its proof) relating a concrete source and target block as part of assignment elimination. This entire snippet is automatically generated and Isabelle successfully checks it (and the remainder of the certificate) automatically. In our generated certificate, the proof of the global block theorem relating the same blocks as part of assignment elimination uses this local block lemma.

2.2.5. Discussion of Transformations in Next Sections

In the remainder of this chapter, we will make the high-level ideas from this section concrete for the different transformations applied by Boogie. We do not present the transformations in the order that Boogie applies them. Instead, we first discuss the details for the three most challenging transformations (cycle elimination in Section 2.4, assignment elimination in Section 2.5, and the final generation of the VC in Section 2.6) and how we formally validate them. Since these transformations all operate solely on control-flow graphs, we first present the formal semantics of control-flow graphs in Section 2.3. After discussing these three most challenging transformations, we discuss the CFG optimisations in Section 2.7. Finally, we present the formal semantics for Boogie ASTs in Section 2.8, followed by an overview of the AST-to-CFG transformation in Section 2.9.

2.3. A Formal Semantics for Boogie

Our automatically generated certificates crucially rely on a formal semantics for Boogie programs. One of our contributions is the first such formal semantics for a significant subset of Boogie programs that is mechanised in Isabelle. Our semantics is based on the Boogie reference manual [1], the presentation of its type system [67], and the Boogie implementation for reference [75]. Our Isabelle formalisation is available online [73].

[1]: Leino (2008), *This is Boogie 2*

[67]: Leino et al. (2010), *A Polymorphic Intermediate Verification Language: Design and Logical Encoding*

[75]: Boogie Developers (n.d.), *Boogie implementation*

[73]: Parthasarathy (2024), *Boogie Semantics and Certificate Metatheory Formalisation*

As discussed in the previous section, Boogie uses two program representations: ASTs and CFGs. In terms of the semantics, the representation affects only the control-flow elements (*e.g.* sequential composition, conditional branching, loops), the rest remains the same (*e.g.* basic commands such as `assert` commands). In this section, we present the semantics of control-flow independent elements and present the semantics of CFGs for the control flow. We will discuss the AST representation and its semantics in Section 2.8.

2.3.1. The Boogie Language

Our supported Boogie subset for *control-flow independent* elements is shown in Figure 2.3. A Boogie program consists of a list of background declarations and a list of procedures.

Background and procedure declarations

Background declarations include axioms, uninterpreted (polymorphic) functions, type constructors, global variables, and constants. The global variables and constants represent the *global data* of a Boogie program. Functions can be polymorphic as indicated by the type parameters $\vec{t}$. Type constructor declarations include the constructor name C and the number of type parameters (*e.g.* `type Field _ _` denotes a type constructor named `Field` that takes two type arguments). Function declarations do not provide a function interpretation (*i.e.* there is no function body) and type constructor declarations do not provide interpretations for the corresponding types. Axioms, defined via Boogie expressions, are used

$$
\begin{aligned}
\mathit{BUnaryOp} \ni \mathit{uop} &::= \; - \mid \texttt{!} \\
\mathit{BBinaryOp} \ni \mathit{bop} &::= \; \texttt{==} \mid \texttt{!=} \mid + \mid - \mid * \mid / \mid \mathsf{mod} \mid \leq \mid < \mid \geq \mid > \mid \texttt{\&\&} \mid \texttt{||} \mid \Rightarrow \mid \Leftrightarrow \\
\mathit{BExpr} \ni e &::= x \mid \mathbf{false} \mid \mathbf{true} \mid i \mid \mathit{uop}(e) \mid e \; \mathit{bop} \; e \mid \mathbf{if}\; e \;\mathbf{then}\; e \;\mathbf{else}\; e \mid f[\vec{\tau}](\vec{e}) \mid \mathbf{old}(e) \mid \\
&\quad\;\; \mathbf{forall}\; x : \tau :: e \mid \mathbf{exists}\; x : \tau :: e \mid \mathbf{forall}\; \langle t \rangle :: e \mid \mathbf{exists}\; \langle t \rangle :: e \\
\mathit{BBasicCmd} \ni c &::= \mathbf{assume}\; e \mid \mathbf{assert}\; e \mid x := e \mid \mathbf{havoc}\; x \\
\mathit{BType} \ni \tau &::= \mathsf{int} \mid \mathsf{bool} \mid C\; \vec{\tau} \mid t \\
\mathit{BBackgroundDecl} \ni \mathit{bgDecl} &::= \mathbf{axiom}\; e \mid \mathbf{function}\; f[\vec{t}](\overrightarrow{x : \tau}) \;\mathbf{returns}\; \tau' \mid \mathbf{type}\; C \; \vec{\_} \mid \\
&\quad\;\; \mathbf{var}\; x : \tau \mid \mathbf{const}\; x : \tau \\
\mathit{BProcedureDecl} \ni \mathit{procDecl} &::= \; \mathbf{procedure}\; p(\overrightarrow{x : \tau}) \;\mathbf{returns}\; (\overrightarrow{y : \tau}) \\
&\quad\;\; \mathbf{requires}\; e \\
&\quad\;\; \mathbf{ensures}\; e \\
&\quad\;\; \{ \overrightarrow{\mathbf{var}\; z : \tau}; \; \mathit{body} \} \\
\mathit{BProg} \ni \mathit{prog} &::= \overrightarrow{\mathit{bgDecl}}; \overrightarrow{\mathit{procDecl}}
\end{aligned}
$$

Figure 2.3: The control-flow independent syntax of our formalised Boogie subset. p (procedure name) and C (type constructor name) denote Boogie identifiers. x and y denote variables. i denotes an integer constant. *body* denotes a procedure body, which is either a CFG or AST over basic commands (CFG and AST control-flow elements are not shown in the figure). Here, procedures have only one precondition and one postcondition for the sake of presentation. Our Isabelle formalisation additionally supports specifying multiple preconditions (resp. postconditions), which Boogie interprets as a single precondition (resp. postcondition) given by the conjunction of the specified preconditions (resp. postconditions).

to constrain the possible function and type interpretations (and constants), as we will make clearer later.[4]

A procedure declaration includes parameter ($\overrightarrow{x : \tau}$), result-variable ($\overrightarrow{y : \tau}$), and local-variable ($\overrightarrow{z : \tau}$) declarations (the *local data*), a pre- and postcondition, and a procedure body (represented as an AST or CFG).[5] We discuss our formalisation of CFGs in Subsection 2.3.2, and will discuss ASTs in a later section (Section 2.8 on page 79).

4: In practice, Boogie allows function bodies that are treated as syntactic sugar for an uninterpreted function with an axiom stating that a function call evaluates to the same value as its body for all function arguments.

5: Source-level procedure specifications also include *modifies clauses*, declaring a set of global variables the procedure may modify, which are required to describe the modular semantics of procedure calls. Since we do not support procedure calls, we need not consider modifies clauses.

Types, expressions, basic commands

We support the primitive types int and bool. Moreover, we support types obtained via declared type constructors, which we call *uninterpreted types*; the sets of values inhabiting such types are constrained only via Boogie axioms and **assume** commands. Moreover, types can contain type variables (for instance, as part of the type signature of polymorphic functions).

Boogie's expression syntax is largely standard. Expressions include variables, Boolean and integer literals, unary and binary expressions, conditional expressions, and function calls $f[\vec{\tau}](\vec{e})$. The arguments $\vec{\tau}$ to a function call $f[\vec{\tau}](\vec{e})$ instantiate the *type* parameters in the corresponding function declaration and are inferred by the type-checker; in our formalisation, type parameters are always explicit. Less standard expressions include old expressions **old**(e) which evaluate the expression e w.r.t. the current local data and the global data as it *was* in the pre-state of the procedure execution. Boogie expressions also include universal and existential *value* quantification (written **forall** $x : \tau :: e$

and **exists** $x : \tau :: e$), as well as universal and existential *type* quantification (written **forall** $\langle t \rangle :: e$ and **exists** $\langle t \rangle :: e$). In the latter, the type variable t is bound in e and quantifies over *closed* Boogie types (*i.e.* types that do not contain any type variables).

Note that Boogie's evaluation of an expression e is *total* if e is well-typed. That is, e evaluates to a value if e is well-typed, as we show in our type soundness result for expressions (Subsection 2.3.6). In particular, division and modulo by 0 are defined to be some unknown but fixed value. Moreover, division by a nonzero integer is defined to be the Euclidean division, since SMT solvers use the Euclidean division in this case: Boogie maps integer divisions in an input program directly to the built-in division used by SMT solvers in the generated verification condition.

Basic commands form the single-steps of executions through a Boogie CFG or AST. In a CFG, sequential composition is implicit in the list of basic commands in a CFG basic block and further control flow (including loops) is prescribed by CFG edges. Boogie's basic commands are assumes, asserts, assignments, and havocs. **havoc** x nondeterministically assigns a value matching the type of variable x to x.

Unsupported features

The main Boogie features *not* supported by our CFG subset are maps, other primitive types such as bitvectors, and procedure calls. Boogie maps can be polymorphic and impredicative, *i.e.* one can define maps that contain themselves in their domain. Giving a semantic model for maps in general is nontrivial. However, it is possible to axiomatise maps in our subset (via Boogie functions and axioms). We discuss maps in more detail as part of future work in Section 2.12. Modelling bitvectors is simpler, although maintaining full automation may require some additional work. Adding support for procedure calls is conceptually simple but requires engineering effort. In fact, Boogie desugars procedure calls into our subset using the procedure's specification.

2.3.2. Boogie CFGs

Our formalisation of a Boogie CFG uses the notion of a *block identifier*: each basic block in the CFG is associated with a unique block identifier (in our formalisation, block identifiers are natural numbers). More concretely, a Boogie CFG G is a triple (b_0, V, E), where b_0 is the block identifier for the entry basic block where executions start, V is a partial mapping from block identifiers to the list of *basic commands* contained in the corresponding basic block, and E represents the edges of G via a partial mapping from block identifiers to a list of block identifiers representing the successors (semantically, execution after a basic block continues via any of its successors nondeterministically). We use convenience functions for the three components of a CFG $G = (b_0, V, E)$: (1) $\mathsf{entry}(G)$ for b_0, (2) $\mathit{cmds}(G, b)$ for $V(b)$ (*i.e.* the commands contained in the basic block identified by b), and (3) $\mathsf{successors}(G, b)$ for $E(b)$ (*i.e.* the successors of the basic block identified by b).

A triple (b_0, V, E) represents an actual CFG G only if the domains of V and E contain the set of block identifiers corresponding to all basic blocks in G, and the range of E contains only block identifiers that occur in the domains of V and E. We explicitly check these constraints in our certificates (they hold for all of the concrete CFGs that appear in our certificates). For instance, when proving a global block theorem *gbt* relating a source block b_S and a target block b_T, we need to ensure that $V(b_S)$ and $V(b_T)$ are both defined, otherwise we cannot use the local block lemma relating the commands contained in b_S and b_T.

2.3.3. Operational Semantics

Values and state model

We formalise values in Boogie using the following algebraic data type:

$$'a\ \mathsf{val} \triangleq \mathsf{IntVal}(\mathsf{int}_{isa}) \mid \mathsf{BoolVal}(\mathsf{bool}_{isa}) \mid \mathsf{AbsVal}('a)$$

There are three kinds of values: integers, Booleans, and *abstract values*. We embed integer and Boolean values as their Isabelle counterparts (*i.e.* in our mechanisation, int_{isa} and bool_{isa} in the above definition are the Isabelle types for integers and Booleans, respectively). Abstract values express the values for uninterpreted types that are obtained via type constructors. Our value definition is parameterised by the carrier type $'a$ for the abstract values. That is, we do not fix up front how abstract values should be represented. Each uninterpreted type is (indirectly) associated with a *non-empty* subset of abstract values via a *type interpretation* map $\mathcal{T}$ from the carrier type $'a$ to (single) uninterpreted types that have no type variables (*i.e.* these types are *closed*); particular interpretations of uninterpreted types can be obtained via different choices of type interpretation $\mathcal{T}$. We will show a concrete instantiation of the carrier type and a corresponding type interpretation in Chapter 3.

One can understand Boogie programs in terms of the sets of possible *executions* through each procedure body. Executions are (as usual) composed of sequences of steps according to the semantics of basic commands and paths through the CFG; these can be finite or infinite (representing a non-terminating execution). A finite execution has one of three outcomes: (1) the execution fails, because an **assert** A command is reached in a state not satisfying assertion A or an exit block of the procedure is reached in a state where the postcondition fails, or (2) the execution stops and *goes to magic*, because an **assume** A command is reached in a state not satisfying A, or (3) the execution succeeds and transitions to a state, because neither of the first two cases occur.[6] The three outcomes are represented formally by the following algebraic data type:

6: For AST representations of the procedure body, one must additionally consider loop invariants. In contrast in CFGs, as we will see in Subsection 2.3.5, loop invariants are represented as **assert** commands at the beginning of the corresponding loop entry point.

$$'a\ \mathsf{outcome} \triangleq \mathsf{F} \mid \mathsf{M} \mid \mathsf{N}('a\ \mathsf{state})$$

where (1) F denotes a *failure outcome*, (2) M denotes a *magic outcome*, and (3) $\mathsf{N}(\sigma)$ denotes a *normal outcome*, where σ is the resulting Boogie state ($'a$ state is the corresponding type representing such states that store Boogie values of type $'a$ val). A Boogie state σ is a triple (os, gs, ls) of partial mappings from variables to values for the old global state os (for the evaluation of old expressions), the (current) global state gs, and the local state ls, respectively.

Expression evaluation

An expression e evaluates to value v in the state σ if the (big-step) judgement $(\mathcal{T}, \Lambda, \mathcal{F}), \Omega \vdash \langle e, \sigma \rangle \Downarrow v$ holds in the *Boogie context* $(\mathcal{T}, \Lambda, \mathcal{F})$ and under the *type substitution* Ω. Analogously, a list of expressions es evaluate to values vs in state σ if the judgement $(\mathcal{T}, \Lambda, \mathcal{F}), \Omega \vdash \langle es, \sigma \rangle [\Downarrow] \ vs$ holds. A Boogie context Γ is a triple $(\mathcal{T}, \Lambda, \mathcal{F})$, where $\mathcal{T}$ is a *type interpretation* (as above), Λ is a *variable context* that is given by a pair (G, L) of type declarations for the global (G) and local (L) data, and $\mathcal{F}$ is a *function interpretation*, which maps each function name to a semantic function mapping a list of types (*i.e.* the type parameter instantiations) and a list of argument values to a return value. We use projection functions for the three components of a Boogie context Γ: $\mathsf{TypeInterp}(\Gamma)$ for the type interpretation, $\mathsf{Vars}(\Gamma)$ for the variable context, and $\mathsf{FunInterp}(\Gamma)$ for the function interpretation. A type substitution Ω maps type variables to types.

The two judgements (for the evaluation of a single expression and a list of expressions) are defined mutually and the corresponding rules are shown in Figure 2.4 (evaluation of a single expression) and Figure 2.5 (evaluation of a list of expressions). The rule for variable lookup is defined in terms of the function $\mathsf{lookup}((G, L), (os, gs, ls), x)$, which returns $ls(x)$ if x belongs to the local data (*i.e.* x is recorded in the type declarations L for the local data) and $gs(x)$ otherwise.[7] This models the fact that local variables can shadow global variables.[8] In the rule for literals, l_e and l_v denote literal expressions and the corresponding literal values, respectively. The rules for value quantification are defined in terms of $\mathsf{typ}_{\mathcal{T}}(v)$, which maps a value v to its type w.r.t. the type interpretation $\mathcal{T}$ for abstract values.

7: (G, L) is a variable context, where G and L are the type declarations for the global and local data, respectively.

8: Note that in our generated certificates, our Isabelle embedding of Boogie programs gives unique names to variables in the presence of shadowing, which simplifies the task of certificate generation. It is future work to show that such an embedding captures an embedding where shadowing is reflected explicitly.

The quantification of types quantifies over every possible closed type (*i.e.* types that do not contain type variables). For example, the following rule expresses when a universal type quantification evaluates to true (the type variable t is bound to the quantified type and may occur in e):

$$\frac{\forall \tau.\ \mathsf{closed}(\tau) \Longrightarrow \Gamma, \Omega(t \mapsto \tau) \vdash \langle e, \sigma \rangle \Downarrow \mathsf{BoolVal}(\mathsf{true})}{\Gamma, \Omega \vdash \langle \mathbf{forall}\ \langle t \rangle :: e, \sigma \rangle \Downarrow \mathsf{BoolVal}(\mathsf{true})}$$

The premise requires one to show that the expression e reduces to true for every possible type τ that is closed.

Type quantification and type declarations

Note that our semantics does not depend on the type constructors declared in a program: the semantics of quantification over types also considers types obtained via undeclared type constructors. As we will show in Section 2.6, the VC generated by Boogie reflects this semantics. Moreover, in Chapter 3, we will show that this semantics is sufficient to formally justify the translation from a Boogie front-end language (*i.e.* Viper) to Boogie.
As an alternative, one could consider a semantics that quantifies only

Unquantified expressions

$$\frac{\mathsf{lookup}(\Lambda, \sigma_v, x) = v}{(\mathcal{T}, \Lambda, \mathcal{F}), \Omega \vdash \langle x, \sigma_v \rangle \Downarrow v} \qquad \frac{}{\Gamma, \Omega \vdash \langle l_e, \sigma \rangle \Downarrow l_v}$$

$$\frac{\Gamma, \Omega \vdash \langle e, \sigma \rangle \Downarrow v' \quad \overline{uop}(v') = v}{\Gamma, \Omega \vdash \langle uop(e), \sigma \rangle \Downarrow v} \qquad \frac{\begin{array}{c}\Gamma, \Omega \vdash \langle e_1, \sigma \rangle \Downarrow v_1 \\ \Gamma, \Omega \vdash \langle e_2, \sigma \rangle \Downarrow v_2 \\ v_1\ \overline{bop}\ v_2 = v\end{array}}{\Gamma, \Omega \vdash \langle e_1\ bop\ e_2, \sigma \rangle \Downarrow v}$$

$$\frac{\begin{array}{c}\Gamma, \Omega \vdash \langle e_1, \sigma \rangle \Downarrow \mathsf{BoolVal(true)} \\ \Gamma, \Omega \vdash \langle e_2, \sigma \rangle \Downarrow v\end{array}}{\Gamma, \Omega \vdash \langle \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3, \sigma \rangle \Downarrow v} \qquad \frac{\begin{array}{c}\Gamma, \Omega \vdash \langle e_1, \sigma \rangle \Downarrow \mathsf{BoolVal(false)} \\ \Gamma, \Omega \vdash \langle e_3, \sigma \rangle \Downarrow v\end{array}}{\Gamma, \Omega \vdash \langle \mathtt{if}\ e_1\ \mathtt{then}\ e_2\ \mathtt{else}\ e_3, \sigma \rangle \Downarrow v}$$

$$\frac{\begin{array}{c}(\mathcal{T}, \Lambda, \mathcal{F}), \Omega \vdash \langle \vec{e}, \sigma \rangle\ [\Downarrow]\ \vec{v}' \\ \mathcal{F}(f) = \overline{f} \quad \overline{f}(\mathsf{map}(\lambda t.\ \mathsf{substT}(\Omega, t), \vec{\tau}), \vec{v}') = v\end{array}}{(\mathcal{T}, \Lambda, \mathcal{F}), \Omega \vdash \langle f[\vec{\tau}](\vec{e}), \sigma \rangle \Downarrow v} \qquad \frac{\Gamma, \Omega \vdash \langle e, (os, os, ls) \rangle \Downarrow v}{\Gamma, \Omega \vdash \langle \mathtt{old}(e), (os, gs, ls) \rangle \Downarrow v}$$

Value quantification

$$\frac{\forall w.\ \mathsf{typ}_{\mathcal{T}}(w) = \mathsf{substT}(\Omega, \tau) \Longrightarrow (\mathcal{T}, (G, L(x \mapsto \tau)), \mathcal{F}), \Omega \vdash \langle e, (os, gs, ls(x \mapsto w)) \rangle \Downarrow \mathsf{BoolVal(true)}}{(\mathcal{T}, (G, L), \mathcal{F}), \Omega \vdash \langle \mathtt{forall}\ x : \tau :: e, (os, gs, ls) \rangle \Downarrow \mathsf{BoolVal(true)}}$$

$$\frac{\mathsf{typ}_{\mathcal{T}}(w) = \mathsf{substT}(\Omega, \tau) \quad (\mathcal{T}, (G, L(x \mapsto \tau)), \mathcal{F}), \Omega \vdash \langle e, (os, gs, ls(x \mapsto w)) \rangle \Downarrow \mathsf{BoolVal(false)}}{(\mathcal{T}, (G, L), \mathcal{F}), \Omega \vdash \langle \mathtt{forall}\ x : \tau :: e, (os, gs, ls) \rangle \Downarrow \mathsf{BoolVal(false)}}$$

$$\frac{\mathsf{typ}_{\mathcal{T}}(w) = \mathsf{substT}(\Omega, \tau) \quad (\mathcal{T}, (G, L(x \mapsto \tau)), \mathcal{F}), \Omega \vdash \langle e, (os, gs, ls(x \mapsto w)) \rangle \Downarrow \mathsf{BoolVal(true)}}{(\mathcal{T}, (G, L), \mathcal{F}), \Omega \vdash \langle \mathtt{exists}\ x : \tau :: e, (os, gs, ls) \rangle \Downarrow \mathsf{BoolVal(true)}}$$

$$\frac{\forall w.\ \mathsf{typ}_{\mathcal{T}}(w) = \mathsf{substT}(\Omega, \tau) \Longrightarrow (\mathcal{T}, (G, L(x \mapsto \tau)), \mathcal{F}), \Omega \vdash \langle e, (os, gs, ls(x \mapsto w)) \rangle \Downarrow \mathsf{BoolVal(false)}}{(\mathcal{T}, (G, L), \mathcal{F}), \Omega \vdash \langle \mathtt{exists}\ x : \tau :: e, (os, gs, ls) \rangle \Downarrow \mathsf{BoolVal(false)}}$$

Type quantification

$$\frac{\forall \tau.\ \mathsf{closed}(\tau) \Longrightarrow \Gamma, \Omega(t \mapsto \tau) \vdash \langle e, \sigma \rangle \Downarrow \mathsf{BoolVal(true)}}{\Gamma, \Omega \vdash \langle \mathtt{forall}\ \langle t \rangle :: e, \sigma \rangle \Downarrow \mathsf{BoolVal(true)}}$$

$$\frac{\mathsf{closed}(\tau) \quad \Gamma, \Omega(t \mapsto \tau) \vdash \langle e, \sigma \rangle \Downarrow \mathsf{BoolVal(false)}}{\Gamma, \Omega \vdash \langle \mathtt{forall}\ \langle t \rangle :: e, \sigma \rangle \Downarrow \mathsf{BoolVal(false)}}$$

$$\frac{\mathsf{closed}(\tau) \quad \Gamma, \Omega(t \mapsto \tau) \vdash \langle e, \sigma \rangle \Downarrow \mathsf{BoolVal(true)}}{\Gamma, \Omega \vdash \langle \mathtt{exists}\ \langle t \rangle :: e, \sigma \rangle \Downarrow \mathsf{BoolVal(true)}}$$

$$\frac{\forall \tau.\ \mathsf{closed}(\tau) \Longrightarrow \Gamma, \Omega(t \mapsto \tau) \vdash \langle e, \sigma \rangle \Downarrow \mathsf{BoolVal(false)}}{\Gamma, \Omega \vdash \langle \mathtt{exists}\ \langle t \rangle :: e, \sigma \rangle \Downarrow \mathsf{BoolVal(false)}}$$

Figure 2.4: Rules for the evaluation of expressions (defined mutually with the evaluation of a list of expressions shown in Figure 2.5). $\overline{uop}$ and $\overline{bop}$ denote the semantic interpretation of a unary operation uop and binary operation bop, respectively. $\mathsf{substT}(\Omega, \tau)$ denotes the substitution of type variables in the type τ according to the type substitution Ω. In our Isabelle mechanisation, we track bound variables differently compared to other variables, which leads to slightly different rules for variable lookups and value quantification, which we ignore here for the sake of presentation.

$$\frac{}{\Gamma, \Omega \vdash \langle [], \sigma \rangle \; [\Downarrow] \; []} \qquad \frac{\Gamma, \Omega \vdash \langle e, \sigma \rangle \Downarrow v \qquad \Gamma, \Omega \vdash \langle es, \sigma \rangle \; [\Downarrow] \; vs}{\Gamma, \Omega \vdash \langle (e :: es), \sigma \rangle \; [\Downarrow] \; (v :: vs)}$$

Figure 2.5: Rules for the evaluation of a list of expressions (defined mutually with the evaluation of a single expression shown in Figure 2.4). The term [] denotes the empty list, and the term $e :: es$ denotes the list whose head and tail are given by e and es, respectively.

over types that could be obtained via declared type constructors. We conjecture that the VC generated by Boogie also respects this semantics (we have not formally proved this). We decided against this alternative for two reasons. First, the alternative would require reasoning about a condition capturing solely the declared type constructors, which could be cumbersome. Second, while the *relevant* types for a program are expressible via the declared type constructors, not all types expressed via declared type constructors are relevant. For example, Boogie front-ends may use a type constructor `type Field _` to express fields for a heap where the type argument expresses the type of values stored at the field. The concrete type `Field (Field int)` is an irrelevant type, since front-ends typically do not allow storing fields into fields. Thus, even with the alternative quantification, which quantifies only over types that could be obtained via declared type constructors, one must still consider irrelevant types.
We conjecture that one can prove a relationship between these two alternatives. For example, if one restricts interpretations in our version (that considers all possible closed types) to collapse all types that cannot be expressed via the declared type constructors to an existing type (*e.g.* the Booleans), then the two alternatives might be *semantically equivalent* w.r.t. the correctness of a Boogie programs (we have not proved this result). This would mean in this case that exchanging one alternative for the other would not affect whether a Boogie program is correct.

In general, expression evaluation is possible only for well-typed expressions; we also formalise Boogie's type system and (for the first time) prove its type soundness for expressions in Isabelle as we discuss in Subsection 2.3.6.

Command and CFG reduction

The (big-step) judgement $\Gamma \vdash \langle c, s \rangle \longrightarrow s'$ defines when a basic command c reduces in outcome s to outcome s' w.r.t. Boogie context Γ.[9] This reduction is lifted to a list of basic commands cs to model the semantics of a single execution through a CFG block via the judgement $\Gamma \vdash \langle cs, s \rangle \; [\longrightarrow] \; s'$. The rules are shown in Figure 2.6 (basic command) and Figure 2.7 (list of basic commands). Assignment reduces only if the value to be assigned has the right type, *i.e.* assignment preserves well-typed states. This condition always holds for well-typed programs, but makes some reasoning easier since it ensures that assignments preserve the well-typedness of states without requiring a well-typedness assumption. The rules for assignment and **havoc** rely on $\mathsf{lookup}_T(\Lambda, x)$ that maps the variable x to its declared

9: In our formalisation, this judgement additionally takes a type substitution as an additional parameter (like the expression evaluation judgement). This type substitution is useful to model type parameters in the procedure signature. Since our certificate generation does not support type parameters in procedure signatures, we omit them here in the semantics for the sake of presentation.

$$\frac{\Gamma, \emptyset \vdash \langle e, \sigma \rangle \Downarrow \texttt{true}}{\Gamma \vdash \langle \texttt{assert}\ e, \mathsf{N}(\sigma) \rangle \longrightarrow \mathsf{N}(\sigma)} \qquad \frac{\Gamma, \emptyset \vdash \langle e, \sigma \rangle \Downarrow \mathsf{BoolVal(false)}}{\Gamma \vdash \langle \texttt{assert}\ e, \mathsf{N}(\sigma) \rangle \longrightarrow \mathsf{F}}$$

$$\frac{\Gamma, \emptyset \vdash \langle e, \sigma \rangle \Downarrow \texttt{true}}{\Gamma \vdash \langle \texttt{assume}\ e, \mathsf{N}(\sigma) \rangle \longrightarrow \mathsf{N}(\sigma)} \qquad \frac{\Gamma, \emptyset \vdash \langle e, \sigma \rangle \Downarrow \mathsf{BoolVal(false)}}{\Gamma \vdash \langle \texttt{assume}\ e, \mathsf{N}(\sigma) \rangle \longrightarrow \mathsf{M}}$$

$$\frac{\begin{array}{c}\Gamma, \emptyset \vdash \langle e, \sigma \rangle \Downarrow v \\ \mathsf{lookup}_T(\Lambda, x) = \mathsf{typ}_{\mathcal{T}}(v) \\ \sigma' = \mathsf{update}(\Lambda, \sigma, x, v)\end{array}}{(\mathcal{T}, \Lambda, \mathcal{F}) \vdash \langle x := e, \mathsf{N}(\sigma) \rangle \longrightarrow \mathsf{N}(\sigma')} \qquad \frac{\begin{array}{c}\mathsf{lookup}_T(\Lambda, x) = \mathsf{typ}_{\mathcal{T}}(v) \\ \sigma' = \mathsf{update}(\Lambda, \sigma, x, v)\end{array}}{(\mathcal{T}, \Lambda, \mathcal{F}) \vdash \langle \texttt{havoc}\ x, \mathsf{N}(\sigma) \rangle \longrightarrow \mathsf{N}(\sigma')}$$

$$\frac{}{\Gamma \vdash \langle c, \mathsf{M} \rangle \longrightarrow \mathsf{M}} \qquad \frac{}{\Gamma \vdash \langle c, \mathsf{F} \rangle \longrightarrow \mathsf{F}}$$

Figure 2.6: Rules for the reduction of basic commands.

$$\frac{}{\Gamma \vdash \langle [], s \rangle \ [\longrightarrow]\ s} \qquad \frac{\begin{array}{c}\Gamma \vdash \langle c, s \rangle \longrightarrow s'' \\ \Gamma \vdash \langle cs, s'' \rangle \ [\longrightarrow]\ s'\end{array}}{\Gamma \vdash \langle (c :: cs), s \rangle \ [\longrightarrow]\ s'}$$

Figure 2.7: Rules for the reduction of lists of basic commands.

type w.r.t. the variable context Λ (if x is recorded in Λ, otherwise $\mathsf{lookup}_T(\Lambda, x)$ would not be defined), and on $\mathsf{update}(\Lambda, \sigma, x, v)$, which returns the state σ where x is updated to v (ensuring that the local state is updated if x is local and otherwise the global state is updated).

The operational semantics of CFGs is modelled by the (small-step) judgement $\Gamma, G \vdash \delta \longrightarrow_{\mathsf{CFG}} \delta'$, expressing that the CFG configuration δ reduces to configuration δ' in the CFG G w.r.t. the Boogie context Γ in a *single step*; the rules are shown in Figure 2.8. A CFG configuration is either *active* or *final*. An active configuration is given by a tuple $(\mathsf{inl}(b), s)$, where b is the block identifier indicating the current position of the execution and s is the current outcome. A final configuration consists of a tuple $(\mathsf{inr}(()), s)$ for outcome s (and unit value $()$) and is reached at the end of a block that has either no successors, or is in a magic or failure outcome.

We derive two judgements from this single-step judgement for expressing an execution that performs multiple reduction steps. First, $\Gamma, G \vdash \delta \longrightarrow^*_{\mathsf{CFG}} \delta'$ denotes the reflexive-transitive closure. That is, δ' is reached from δ via zero or more reduction steps. Second, $\Gamma, G \vdash \delta \longrightarrow^i_{\mathsf{CFG}} \delta'$ expresses that δ' is reached from δ in precisely i reduction steps ($i \in \mathbb{N}$).

2.3.4. Procedure Correctness

A procedure is *correct* if for any *well-formed* type and function interpretations, the procedure body has *no failing executions* in any state σ that (1) satisfies the axioms when restricted to its constants (*i.e.* the state σ where only the constants are retained satisfies the axioms),[10] (2) satisfies the precondition, and (3) is well-typed w.r.t. the variable declarations. This is a *partial correctness* semantics; a procedure body whose executions

10: Boogie does not allow global variables in axioms

$$\frac{cmds(G,b) = cs \quad b' \in \mathsf{successors}(G,b) \quad \Gamma \vdash \langle cs, \mathsf{N}(\sigma)\rangle\ [\rightarrow]\ \mathsf{N}(\sigma')}{\Gamma, G \vdash (\mathsf{inl}(b), \mathsf{N}(\sigma)) \rightarrow_{\mathsf{CFG}} (\mathsf{inl}(b'), \mathsf{N}(\sigma'))} \qquad \frac{cmds(G,b) = cs \quad \mathsf{successors}(G,b) = \emptyset \quad \Gamma \vdash \langle cs, \mathsf{N}(\sigma)\rangle\ [\rightarrow]\ \mathsf{N}(\sigma')}{\Gamma, G \vdash (\mathsf{inl}(b), \mathsf{N}(\sigma)) \rightarrow_{\mathsf{CFG}} (\mathsf{inr}(()), \mathsf{N}(\sigma'))}$$

$$\frac{cmds(G,b) = cs \quad \Gamma \vdash \langle cs, \mathsf{N}(\sigma)\rangle\ [\rightarrow]\ \mathsf{M}}{\Gamma, G \vdash (\mathsf{inl}(b), \mathsf{N}(\sigma)) \rightarrow_{\mathsf{CFG}} (\mathsf{inr}(()), \mathsf{M})} \qquad \frac{cmds(G,b) = cs \quad \Gamma \vdash \langle cs, \mathsf{N}(\sigma)\rangle\ [\rightarrow]\ \mathsf{F}}{\Gamma, G \vdash (\mathsf{inl}(b), \mathsf{N}(\sigma)) \rightarrow_{\mathsf{CFG}} (\mathsf{inr}(()), \mathsf{F})}$$

Figure 2.8: Rules for the reduction of CFGs.

never leave a loop is trivially correct provided that no intermediate `assert` commands fail.

The following formal correctness definition for a Boogie procedure p w.r.t. global declarations *decls* reflects this notion of correctness directly (*decls* includes declarations of functions, axioms, global variables and constants):

Definition 2.3.1 (Correctness of a procedure)

$$\mathit{procCorrect}(decls, p) \triangleq \forall \mathcal{T}, \mathcal{F}, gs, ls.$$

$$\begin{pmatrix} (\forall t.\ \mathsf{closed}(t) \Longrightarrow \exists v.\ \mathsf{typ}_{\mathcal{T}}(v) = t) \land \\ \mathsf{funInterpWellTy}(\mathcal{T}, \mathsf{functions}(decls), \mathcal{F}) \land \\ \mathsf{varMappingWellTy}(\mathcal{T}, \mathsf{consts}(decls)@\mathsf{globals}(decls), p, gs, ls) \land \\ \mathsf{axiomSat}(\mathcal{T}, \mathcal{F}, \mathsf{consts}(decls), \mathsf{axioms}(decls), gs) \end{pmatrix} \Longrightarrow$$

$$\mathit{let}\ \sigma_0 = (gs, gs, ls)\ \mathit{in}$$

$$\mathit{let}\ \Lambda = (\mathsf{consts}(decls)@\mathsf{globals}(decls), \mathsf{params}(p)@\mathsf{results}(p)@\mathsf{locals}(p))\ \mathit{in}$$

$$\mathit{let}\ \Gamma = (\mathcal{T}, \Lambda, \mathcal{F})\ \mathit{in}$$

$$\Gamma, \emptyset \vdash \langle \mathsf{pre}(p), \mathit{N}(\sigma_0)\rangle \Downarrow \mathit{BoolVal}(\mathit{true}) \Longrightarrow$$

$$\mathit{bodyCorrect}(\Gamma, \mathsf{body}(p), \mathsf{post}(p), \sigma_0)$$

The first two conjuncts in the left-hand side formalise when the (universally quantified) type interpretation $\mathcal{T}$ and function interpretation $\mathcal{F}$ are well-formed: The type interpretation must inhabit every uninterpreted closed type and the function interpretation must be well-typed w.r.t. the declared function signatures (*e.g.* every declared function must have an interpretation that given arguments of the declared argument types returns a value of the declared return type). We require the well-formedness of type interpretations in our validation of VC generation, as we will discuss in Subsection 2.6.3 on page 66. We require the well-formedness of function interpretations to prove type soundness, as we will discuss in Subsection 2.3.6 on page 35.

The definition universally quantifies over the global state gs and the local state ls, which are constrained to be well-typed w.r.t. the variable declarations (via varMappingWellTy). Moreover, gs is constrained to satisfy the axioms when restricted to the constants (via axiomSat).[11] The initial state σ_0 is then constructed via these components where the old global state matches the global state, since they are the same at the beginning of a procedure execution.

11: Note that there is no relationship between the local state and the axioms, since the axioms cannot refer to local variables.

In the definition, Γ is the Boogie context that is used for the judgements in the operational semantics. Its variable context Λ is constructed directly via the constant and global variable declarations for the global data, and all the variable declarations associated with the procedure for the local data.

Finally, the conclusion $\mathsf{bodyCorrect}(\Gamma, \mathsf{body}(p), \mathsf{post}(p), \sigma_0)$ expresses that the body of procedure p has no failing executions when starting in the state σ_0 w.r.t. context Γ. This includes that the postcondition $\mathsf{post}(p)$ is satisfied whenever the procedure body finishes its execution. Since the body can be represented either via a CFG or an AST, bodyCorrect is a parameter of our procedure correctness definition, which we instantiate in two separate ways for the two representations. We will discuss the AST instantiation for bodyCorrect in Section 2.8. The CFG instantiation for bodyCorrect is given by:

Definition 2.3.2 (Correctness of a CFG body)

$$\begin{aligned}
&\mathit{bodyCorrect}_{\mathit{CFG}}(\Gamma, G, \mathit{post}, \sigma) \triangleq \\
&\forall r, s'.\ \Gamma, G \vdash (\mathit{inl}(\mathit{entry}(G)), \mathit{N}(\sigma)) \rightarrow^{*}_{\mathit{CFG}} (r, s') \Rightarrow \\
&\quad s' \neq \mathit{F} \wedge \\
&\quad (r = \mathit{inr}(()) \Rightarrow \forall \sigma'.\ s' = \mathit{N}(\sigma') \Rightarrow \Gamma, \emptyset \vdash \langle \mathit{post}, \mathit{N}(\sigma') \rangle \Downarrow \mathit{BoolVal}(\mathit{true}))
\end{aligned}$$

where G is a CFG and $\mathsf{entry}(G)$ is the entry block of G.

The postcondition must be satisfied only if a final configuration is reached normally, while failing states must be unreachable.

2.3.5. Boogie Program Examples

We now illustrate two Boogie programs to make our formal semantics more intuitive. The first one focuses on the procedure body and control flow, while the second one focuses on global declarations and Boogie's type system. We will use the first example as a running example for the remainder of this chapter.

Running example

Figure 2.9 shows a procedure in source code (on the left) and the CFG representation of its body (on the right). `i` and `j` are local variables and the pre- and postconditions are trivial (*i.e.* `true`). We assume that there are no background declarations. We will use the CFG representation as a running example for the transformations whose source procedure is represented as a CFG.

The source code has a while-loop with a classical Floyd-Hoare-style inductive invariant. The invariant is represented implicitly in the CFG representation via an `assert` command at the loop head block B_1. This captures the fact that the invariant must hold on entry of the loop and at the beginning and end of every loop iteration. The CFG has infinite executions: those which start from any state in which `i` is negative. Executions starting from a state in which `i` is zero go to magic due to the initial `assume` command in block B_0; they do not reach the loop.

```
procedure procRunning()
  requires true;
  ensures  true;
{
  var i: int;
  var j: int;
  assume i != 0;
  j := 0;
  while(i != 0)
    invariant
      j >= 0 && (i == 0 ⇒ j > 0);
  {
    if(i < 5)
    {
      j := j+1;
    }
    i := i-1;
  }
  assert j > 0;
}
```

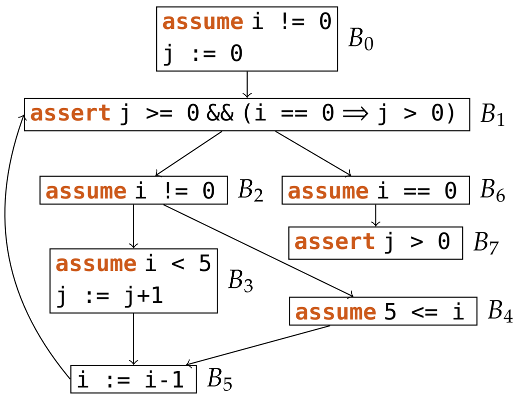


Figure 2.9: Running example procedure shown on the left and the CFG representation of its body shown on the right.

The procedure is correct (*i.e.* has no failing executions): all other initial states will result in executions that satisfy the loop invariant and the final `assert` command. If we removed the initial `assume` in block B_0, however, there *would* be failing executions: the loop invariant check would fail if `i` were initially zero.

Example with background declarations

Figure 2.10 shows a Boogie program with background declarations, polymorphism, universal type quantification, and universal value quantification. The program declares two type constructors: `ref` and `List` to represent references and lists, respectively. The first type argument for `List` is intended to reflect the values stored in the list. Recall that type constructors in Boogie do not provide any interpretation for the types; the declared functions and axioms restrict the possible interpretations, and correctness of a Boogie program must be guaranteed under any well-formed type and function interpretation.

In Figure 2.10, the intention for the polymorphic function `elem` is to check whether an element is in a list, and the intention for the polymorphic function `cons` is to prepend an element to a list. As for the type constructors, these function declarations on their own do not define a function interpretation. The two axioms in Figure 2.10 restrict the possible interpretations by expressing when `elem` holds if the list is constructed via `cons`. Here, universal type quantification is used to express a condition on any possible type of values contained in the list. These two axioms on their own do not force `cons` to model prepending an element to a list; more axioms would be needed. In particular, the axioms allow `cons` to add the

```
type ref;
type List _;

const null: ref;

function elem<T>(x: T, xs: List T) : bool;
function cons<T>(x: T, xs: List T) : List T;

axiom (forall <T> :: (forall x: T, xs: List T :: elem(x, cons(x, xs))));
axiom (forall <T> :: (forall x: T, y: T, xs: List T :: x != y ⇒
  elem(x, cons(y, xs)) == elem(x, xs)));

procedure p(xs: List ref) returns (ys: List ref)
    requires !elem(null, xs);
    ensures  !elem(null, ys);
    ensures  (forall r: ref :: elem(r, xs) ⇒ elem(r, ys));
{
    var y: ref;
    var z: ref;
    assume y != null;
    assume z != null;
    ys := cons(y, cons(z, xs));
}
```

Figure 2.10: Boogie program illustrating the use of background declarations and Boogie's type system. Note that the procedure has two postcondition clauses, which is semantically the same as conjoining both clauses.

element anywhere in the list (not necessarily at the beginning). For the correctness of this program, the two declared axioms are sufficient.

The specification of procedure `p` in Figure 2.10 expresses that if the input list does not contain the reference `null` (which is modelled via a constant declaration), then neither does the output list and the elements of the input list are a subset of those of the output list. The procedure body of `p` satisfies this specification by cons-ing two nondeterministically chosen references that are guaranteed to be non-null (the **`assume`** commands ensure that the values are non-null).

The Boogie program in Figure 2.10 is correct. Both axioms are necessary to ensure correctness. For instance, if the first axiom were omitted, then the correctness of the Boogie program would also have to consider type and function interpretations where `elem(y,xs)` holds (where `y` matches the value provided by the local variable declaration in procedure `p`) but `elem(y, cons(y,cons(z,xs)))` does not. As a result, there would be type and function interpretations under which the postcondition is not guaranteed to hold, and thus the Boogie program would not be correct.

2.3.6. Type Soundness of Expressions

For some of our generated certificates, we require that *type soundness* holds for Boogie expressions. That is, if a Boogie expression e is well-typed and has type τ, then e evaluates to some value of type τ. Boogie's type system for expressions has been presented on paper before [67], but type soundness has not been formally proved before. We formally prove

[67]: Leino et al. (2010), *A Polymorphic Intermediate Verification Language: Design and Logical Encoding*

type soundness of Boogie expressions in Isabelle. Moreover, we develop an approach to automatically prove in Isabelle that a concrete expression e is well-typed such that we can then use our type soundness result to obtain that e reduces to a value of the corresponding type.

We formally define the typing judgement $F, \Delta \vdash e : \tau$, which expresses that e has type τ w.r.t. function declarations F and a *type environment* Δ (a partial mapping from variable names to types). The type soundness theorem that we prove is given by:

Theorem 2.3.1 (Type Soundness of Expressions) *Let* $(\mathcal{T}, \Lambda, \mathcal{F})$ *be an arbitrary Boogie context and let* Ω *be an arbitrary type substitution. Then, if*

1. $F, (\lambda x.\ \mathsf{lookup}_T(\Lambda, x)) \vdash e : \tau$
2. $\mathsf{funInterpWellTy}(\mathcal{T}, F, \mathcal{F})$
3. $\mathit{wfTyFunDecls}(F)$
4. $\mathit{wfTyVarCtxt}(\Omega, \Lambda)$
5. $\mathit{wfTyExpr}(\Omega, e)$
6. $\mathit{stateWellTy}(\mathcal{T}, \Lambda, \Omega, \sigma)$

then: $\exists v.\ (\mathcal{T}, \Lambda, \mathcal{F}), \Omega \vdash \langle e, \sigma \rangle \Downarrow v \wedge \mathsf{typ}_{\mathcal{T}}(v) = \mathsf{substT}(\Omega, \tau)$

The first assumption is the typing assumption, where the type environment is derived from the variable context Λ. The other five assumptions ensure that (1) the function interpretation $\mathcal{F}$ respects the types in the function declarations (via funInterpWellTy), (2) specified types in a function declaration contain only type variables specified by the declaration (via wfTyFunDecls), (3) the variable context Λ contains only type variables that appear in the domain of the type substitution Ω (via wfTyVarCtxt), (4) the expression e contains only type variables that appear in the domain of the type substitution Ω (via wfTyExpr), and (5) the state σ respects the types declared in the variable context (via stateWellTy). The conclusion states the type soundness result, where the type of the value that the expression reduces to is given by $\mathsf{substT}(\Omega, \tau)$, which denotes the application of the type substitution Ω to type τ.

Our rules for defining the typing judgement $F, \Delta \vdash e : \tau$ conceptually match those from Leino and Rümmer [67]. All rules are standard except for the typing rule for equality and disequality given by:

[67]: Leino et al. (2010), *A Polymorphic Intermediate Verification Language: Design and Logical Encoding*

$$\frac{\mathit{bop} \in \{==, !=\} \quad F, \Delta \vdash e_1 : \tau_1 \quad F, \Delta \vdash e_2 : \tau_2 \quad \mathsf{substT}(\Omega, \tau_1) = \mathsf{substT}(\Omega, \tau_2)}{F, \Delta \vdash e_1\ \mathit{bop}\ e_2 : \mathsf{bool}}$$

So, e_1 and e_2 need not have the same type in order for their (dis)equality to be well-typed. Instead, e_1 and e_2 need to have types τ_1 and τ_2, respectively, such that there is *some* type substitution Ω under which τ_1 and τ_2 agree.

The motivation for this rule is to be able to type expressions such as:[12]

12: This particular expression pattern is used in Boogie programs generated by Viper.

$$\texttt{forall}\ \langle t \rangle :: \texttt{forall}\ f : \texttt{Field}\ t :: f\ \texttt{!=}\ g \Rightarrow e$$

where g has, for instance, type `Field int`. Here f `!=` g is well-typed, because, the type substitution $[t \mapsto \mathsf{int}]$ serves as a witness for the above rule. Note that *the semantics* of type quantification also considers instantiations of t for which $t \neq \mathsf{int}$ and thus f and g do not have the same type. In such a case f `!=` g evaluates to true in the semantics, since values of different types are always different.

The rule for (dis)equality is the main challenge for the automation of well-typedness proofs for concrete Boogie expressions, since one must provide a correct type substitution for each (dis)equality. We solve this in our instrumented Boogie verifier implementation by providing a hint that has the same structure as the Boogie expression and where the nodes corresponding to (dis)equalities contain the type variable substitution witnesses. We are able to extract these witnesses directly from Boogie's type inference implementation via a lightweight instrumentation of the existing Boogie verifier implementation. In our generated certificates, we provide these hints to an Isabelle tactic that we developed, which automatically proves that an expression is well-typed.

2.4. Cycle Elimination

In this section, we present our certificate generation approach for the transformation subsequence in the Boogie verifier consisting of the cycle elimination transformation, the pre- and postcondition insertion transformation, and the empty block insertion transformation (this is the certificate connecting procedure P_3 with procedure P_6 in Figure 2.1 on page 17). For the sake of presentation, we will first focus on cycle elimination, which is by far the most challenging among the three transformations, and then at the end discuss how we include the other two transformations. Cycle elimination is challenging as it substantially changes the executions in the CFG (going from looping to non-looping executions), inserts additional nondeterministic assignments and **`assume`** commands, and must do so correctly for arbitrary (*reducible*) nested loop structures, which can include unstructured control flow (*e.g.* jumps out of loops).

2.4.1. Cycle Elimination Overview

Cycle elimination applies to every *loop head* block identified by Boogie's implementation and any *back edges* (following standard definitions for *reducible* CFGs [76, 77]). Intuitively, a loop head is the entry block for a loop and a back edge is an edge from a block within a loop *back* to a corresponding loop head. Figure 2.11 illustrates the transformation's effect on our running example. Block B_1 is the only loop head here, and the edge from B_5 to it is the only back edge (completing looping paths via B_2 and B_3 or B_2 and B_4). An **`assert`** A statement starting a loop head (like B_1) is interpreted as declaring A to be the loop invariant.[13] Cycle elimination performs the following four steps:

[76]: Hecht et al. (1972), *Flow Graph Reducibility*
[77]: Hecht et al. (1974), *Characterizations of Reducible Flow Graphs*

13: In general, multiple asserts at the beginning of a loop head may form the invariant (in such a case, the conjunction of the corresponding expressions forms the invariant).

1. Accumulate a set X_H of all (local and global) variables *assigned-to* on *any looping path* from the loop head back to itself. In our example, X_H is `{i,j}`.
2. Move the **`assert`** A statement declaring a loop invariant (if any) from the loop head to the end of *each preceding* block (in our example: B_0 and B_5).

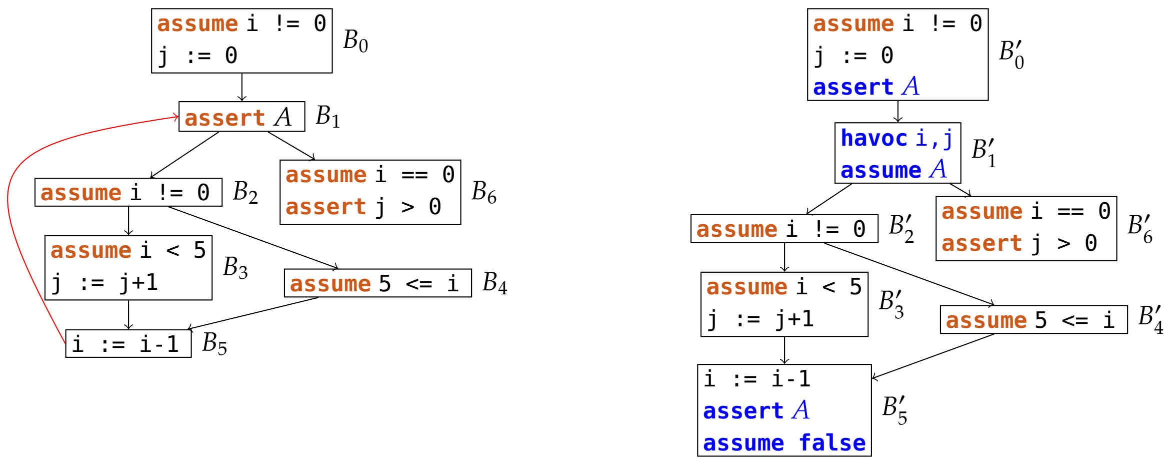


Figure 2.11: Cycle elimination applied to the running example (source is left, target is right). The back edge (the red edge from B_5 to B_1 in the left CFG) is eliminated. The blue commands are new. A is given by `j >= 0 && (i == 0` $\Rightarrow$ `j > 0)`. Note that the blocks B_6 and B_7 from the original CFG representation in Figure 2.9 on page 34 are coalesced to B_6 here due to the CFG optimisations that occur before cycle elimination.

3. Insert **`havoc`** commands at the start of the loop head block per variable in X_H, followed by a single **`assume`** A statement (preceding any further statements).
4. For each block with a back edge to a loop head, delete the back edge; if this leaves the block with no successors, append **`assume false`** to its commands.[14]

14: Omitting **`assume false`** if there are no successors would be incomplete, since otherwise there could be executions reaching the end of the block in a normal outcome in which case the postcondition would have to be satisfied.

The target CFG constructed by cycle elimination overapproximates the set of looping executions of the source CFG via the loop invariant. The **`assert`** command added in step 2 to blocks that are not part of the loop itself (*e.g.* B_0) ensures that the loop invariant holds right before entering the loop. The havoc-then-assume sequence introduced in step 3 can be understood as generating executions for *arbitrary values of* X_H satisfying the loop invariant A, effectively overapproximating the set of states reachable at the loop head in the original program (variables not in X_H are not affected by the loop). In particular, the remnants of any originally looping path (*e.g.* $B'_1 \to B'_2 \to B'_3 \to B'_5$) enforce that any non-failing execution starting from any such state must result in a state which re-establishes the loop invariant before going back to the loop head (*e.g.* due to the **`assert`** added to the origin of a back edge such as B'_5 in step 2). Such paths exist only to ensure that the loop invariant is preserved by every loop iteration (analogously to the premise of a Hoare logic while rule) and to ensure that no loop iteration fails.

The parts of an execution in the source CFG *after* leaving the loop are captured in the target CFG by an execution that does not go through *any* complete iteration of the loop. For example, in Figure 2.11, an execution in the source CFG that leaves the loop from B_1 to B_6 (after potentially multiple executions through the loop) is captured from B_6 onwards by an execution in the target CFG via an execution that goes from B'_1 to B'_6 (without visiting any blocks within the loop). Intuitively, such a capturing execution from B'_1 to B'_6 must exist since the loop invariant overapproximates the possible set of states reachable at the loop head (if all the introduced **`assert`** commands in the target CFG always succeed).

Note that here a single source execution that executes multiple loop iterations is generally not captured by a single target execution but by multiple target executions. Each loop iteration in the same source execution is potentially captured by a different target execution.

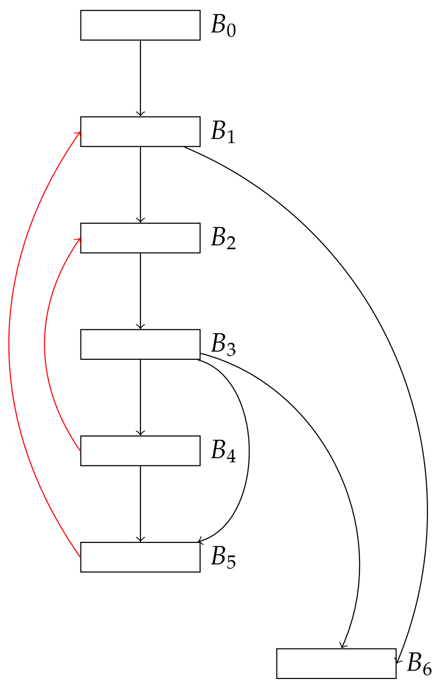


Figure 2.12: A reducible CFG with two loops, where the loop with loop head B_2 is nested within the loop with loop head B_1. The back edges are shown in red.

The above intuition for the soundness of cycle elimination must be formally justified. In particular, this must be achieved not only for the simple CFG in Figure 2.11, but for any reducible looping structure. In general, a loop head may have multiple back edges, looping structures may nest, and edges may exit multiple loops. For example, Figure 2.12 shows a more complex nesting structure with two loops, one nested within the other. In this example, there are three edges that exit the inner loop. On the one hand, the edges from B_3 to B_5 and from B_4 to B_5 both go to the outer loop. On the other hand, the edge from B_3 to B_6 goes to an exit block of the CFG, thus exiting the inner *and* the outer loop.

In general, for cycle elimination to be sound, the CFG must be reducible (as we make clear below), and Boogie must identify the loop heads and corresponding back edges accurately, which is complex in general. Importantly (but perhaps surprisingly), our work makes this transformation of Boogie certifying *without explicitly* checking whether the CFG is reducible or whether the removed edge is indeed a back edge (or even defining these notions). Before we show our certification strategy, we give an intuition for what can go wrong when the source CFG is irreducible.

2.4.2. The Need for Reducibility

Intuitively, a CFG is reducible if there is no loop that has multiple entry points. That is, each loop has a (unique) loop head H that *dominates* all blocks in the corresponding loop, which means that any path from the entry block of the CFG to a block in the loop must first go through H. In particular, this means that in a reducible CFG a loop head H dominates

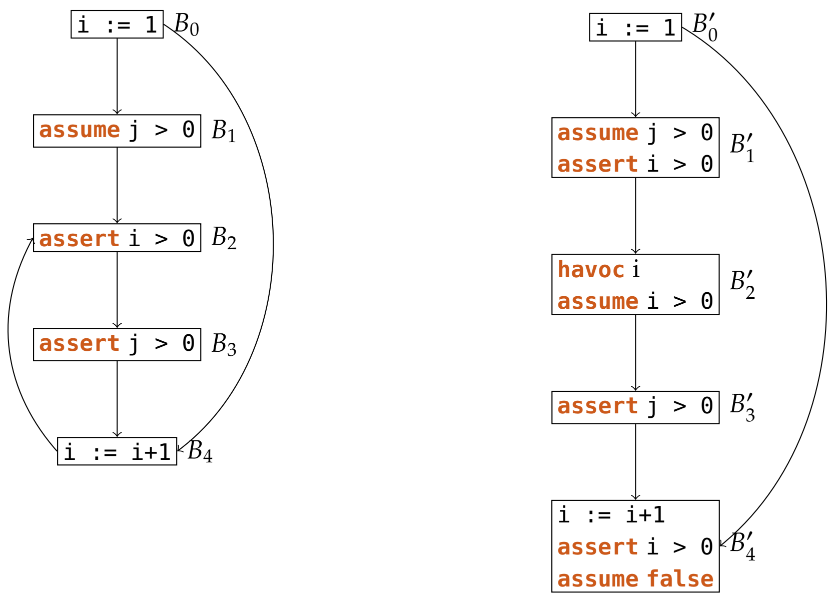


Figure 2.13: An example where the source CFG on the left is irreducible and the resulting target CFG after cycle elimination is shown on the right (if the edge from B_4 to B_2 is eliminated and thus the invariant `i > 0` is used). Here, the transformation is unsound. Boogie rejects the program, since the source CFG is irreducible.

the origin block of each corresponding back edge that leads back to H. The CFGs in Figure 2.11 and Figure 2.12 are reducible. For the CFG in Figure 2.12, the unique entry point of the outer loop is B_1 and the unique entry point of the inner loop is B_2.

The example in Figure 2.13 shows why the cycle elimination transformation presented above would be unsound in general if the source CFG is irreducible. Here, the source CFG (shown on the left) is irreducible, because the loop has two entry points: B_2 and B_4. The target CFG (shown on the right) shows the effect of cycle elimination if one identifies B_2 as the loop head and thus eliminates the edge from B_4 to B_2. This leads to an unsound transformation because the source CFG has a failing execution and the target CFG does not. For the source CFG, the execution e along the path $B_0 \longrightarrow B_4 \longrightarrow B_2 \longrightarrow B_3$ fails in an initial state where `j` evaluates to 0, since `j > 0` does not hold when B_3 is reached. However, the target CFG has no failing executions. In particular, every execution that reaches B'_3 must have gone through B'_1, which prunes all executions where `j > 0` does not hold. Since `j` is not modified by the loop, `j` is not havocked in the target CFG and, as a result, the **`assert`** in B'_3 succeeds. However, in the source CFG, not all executions (*e.g.* the execution e) entering the loop from B_4 satisfy `j > 0` at block B_3.

As the example shows, the high-level problem with multiple loop entry points is that the different sets of executions entering the loop through different entry points may satisfy different constraints on variables not modified by the loop. As a result, if one eliminates edges going back to one of the entry points, then one captures only those executions going through that entry point. Therefore, it is crucial that the CFG is reducible such that every loop has a unique entry point and thus that the identified entry point (*i.e.* the loop head) dominates every block in the loop.

2.4.3. Local Block Lemmas

For the certification of cycle elimination, we follow the general strategy discussed in Subsection 2.2.3 on page 20. The first step is to define and prove the local block lemmas relating a source block and the corresponding target block. Recall that these imply that if executing the statements of a target block yields no failing executions, the same holds for the corresponding source block; this result is trivial for source blocks other than loop heads and their immediate predecessors, since these are left unchanged by cycle elimination. To enable eventual composition of our local block lemmas (to prove the global block theorems), we need to also reflect the role of the `assume` and `assert` commands employed in this transformation. The formal statement of our local block lemmas is as follows:[15]

15: We present our local block lemmas (and global block theorems) as "theorems" in this dissertation. Note that these results are proved on each run of the verifier for concrete blocks; this is not a single theorem that we show once and for all.

Theorem 2.4.1 (Cycle elimination local block lemma) *Let B be a source block with commands cs_S, whose corresponding target block has commands cs_T. If B is a loop head, let X_H be as defined in cycle elimination step 1 (and the empty set otherwise) and let A_{pre} be its loop invariant (and* `true` *otherwise). If B is a* predecessor *of a loop head, let A_{post} be the loop invariant of its successor (and* `true` *otherwise). Let Λ be the variable context for the source and target procedures. Then, for any type interpretation $\mathcal{T}$, any well-formed function interpretation $\mathcal{F}$ w.r.t. $\mathcal{T}$, any states σ_1 and σ_2, and any outcome s'_1,* if*:*

1. $(\mathcal{T}, \Lambda, \mathcal{F}) \vdash \langle cs_S, \mathsf{N}(\sigma_1)\rangle \, [\rightarrow] \, s'_1$
2. $\forall s'_2.\ (\mathcal{T}, \Lambda, \mathcal{F}) \vdash \langle cs_T, \mathsf{N}(\sigma_2)\rangle \, [\rightarrow] \, s'_2 \Longrightarrow s'_2 \neq \mathsf{F}$
3. *A_{pre} is satisfied in σ_1, and σ_2* matches *σ_1 on all variables* w.r.t. variable context *Λ except for those in X_H (see below for what it means for states to match w.r.t. a variable context)*
4. *σ_1 and σ_2 are well-typed w.r.t. the variable context Λ where the types are interpreted wrt $\mathcal{T}$ (i.e. $\mathsf{stateWellTy}(\mathcal{T}, \Lambda, \emptyset, \sigma_i)$ for $i \in \{1, 2\}$)*

then*: $s'_1 \neq \mathsf{F}$, and if s'_1 is a normal outcome $\mathsf{N}(\sigma'_1)$, then (1) A_{post} is satisfied in σ'_1, (2) $\mathsf{stateWellTy}(\mathcal{T}, \Lambda, \emptyset, \sigma'_1)$ holds, and (3) if no* `assume false` *was added at the end of cs_T, then there is a target execution in cs_T from $\mathsf{N}(\sigma_2)$ that reaches a normal outcome $\mathsf{N}(\sigma'_2)$ whose state matches σ'_1 on all variables w.r.t. Λ, and $\mathsf{stateWellTy}(\mathcal{T}, \Lambda, \emptyset, \sigma'_2)$ holds.*

Note that two states σ_1 and σ_2 *match w.r.t. a variable context* Λ on a variable x if and only if: if x is a *local* variable according to Λ, then the *local* state components of σ_1 and σ_2 store the same value for x, and otherwise the *global* state components of σ_1 and σ_2 store the same value for x.

The gist of this lemma is to capture *locally* the ideas behind the four steps of the transformation. For example, consequence (1) reflects that *after* the transformation, any blocks that *were previously* predecessors of a loop head (B'_0 and B'_5 in our running example) will have an `assert` command checking for the corresponding invariant. So if the target program has no failing executions, in each source execution this invariant will be true at that point.

The third assumption reflects that loop heads in the target CFG include a `havoc` of the modified loop variables X_H followed by the `assume` of the loop invariant. The idea is that if cycle elimination is sound, then for any source execution that reaches the loop head in a state $\mathsf{N}(\sigma_1)$ (via potentially multiple loop iterations), it must hold that (R1) the loop invariant holds in σ_1, and (R2) there is a corresponding execution in the target CFG that reaches the corresponding block without executing loop iterations in a state $\mathsf{N}(\sigma_2)$ that agrees with σ_1 on all variables different from X_H. In the case of loop heads, the local block lemma will be applied only for such states σ_1 and σ_2. The third assumption captures (R1) directly in order to justify the inserted **`assume`** command in the target block, and further requires that the two states differ only in the values of variables in X_H, which is required to ensure that *after* executing the **`havoc`** commands in the target block, the two states are again fully in sync as guaranteed by conclusion (3) of the local block lemma.

The fourth assumption requires that the two states are well-typed. We need this assumption together with the constraint that the function interpretation is well-formed to prove that the inserted **`assert`** commands, which check the invariants, reduce. The reason is that the loop invariant is not necessarily evaluated in a corresponding state in the source block. Thus, we cannot use the first assumption (the existence of a source execution through the source block) to prove that the loop invariant evaluates to a Boolean and thus prove that the inserted **`assert`** command in the target block reduces.[16] Instead, we prove the inserted **`assert`** command reduces by showing that the loop invariant I evaluates to a Boolean in the target state. We achieve this by proving that I is well-typed and then use the well-typedness assumption (to conclude that the target state is well-typed) and our type soundness result (see Subsection 2.3.6 on page 35).

16: For the inserted **`assume`** command, we can use assumption 3.

2.4.4. Generating Proofs for Local Block Lemmas

To automatically generate a proof for a local block lemma that can be automatically checked by Isabelle, we define a relation on the lists of basic commands in CFG blocks, which captures the possible syntactic relationships between a source CFG block and target CFG block for cycle elimination based on different cases (*e.g.* whether the source block is a loop head or whether a successor block is a loop head). More precisely, we define a relation between source block commands cs_S and target block commands cs_T of the following form (whose definition we will show shortly):

$$\mathsf{cycleElimRel}(X_H^{list}, A_{pre}, A_{post}, cut, cs_S, cs_T) \tag{2.1}$$

We prove once and for all that if the parameter cut is a Boolean expressing whether an **`assume false`** is added at the end of the target block, X_H^{list} is a list representation of X_H defined in Theorem 2.4.1 (the local block lemma) and the other parameters are the same as in Theorem 2.4.1, and certain typing properties hold, then the local block lemma holds. Thus, to generate a proof of a local block lemma, we just need to generate proofs for proposition 2.1 and for the required typing properties.

The relation $\mathsf{cycleElimRel}$ is defined inductively. Figure 2.14 shows a simplified version of the rules.[17] The rules syntactically reflect the insertion of statements in the cycle elimination transformation. For example, to derive proposition 2.1 where the source block is a loop head whose

17: Here, we ignore the fact that there can be multiple invariants (their conjunction forms the intended invariant) and that **`assume`** commands at the beginning of a loop head are treated by Boogie as *free invariants*. We handle these aspects in our mechanisation.

$$\frac{\mathsf{cycleElimRel}(X_H^{list}, A_{pre}, A_{post}, cut, cs_S, cs_T)}{\mathsf{cycleElimRel}(x :: X_H^{list}, A_{pre}, A_{post}, cut, cs_S, \texttt{havoc}\ x :: cs_T)}\ (\textsc{havoc-ce})$$

$$\frac{\mathsf{cycleElimRel}([], \texttt{true}, A_{post}, cut, cs_S, cs_T)}{\mathsf{cycleElimRel}([], A_{pre}, A_{post}, cut, \texttt{assert}\ A_{pre} :: cs_S, \texttt{assume}\ A_{pre} :: cs_T)}\ (\textsc{preinv-ce})$$

$$\frac{\mathsf{cycleElimRel}([], \texttt{true}, A_{post}, cut, cs_S, cs_T)}{\mathsf{cycleElimRel}([], \texttt{true}, A_{post}, cut, c :: cs_S, c :: cs_T)}\ (\textsc{id-ce})$$

$$\frac{\mathsf{cycleElimRel}([], \texttt{true}, \texttt{true}, cut, [], cs_T)}{\mathsf{cycleElimRel}([], \texttt{true}, A_{post}, cut, [], \texttt{assert}\ A_{pre} :: cs_T)}\ (\textsc{postinv-ce})$$

$$\frac{}{\mathsf{cycleElimRel}([], \texttt{true}, \texttt{true}, cut, [], [])}\ (\textsc{nil-ce})$$

$$\frac{cut}{\mathsf{cycleElimRel}([], \texttt{true}, \texttt{true}, cut, [], [\texttt{assume false}])}\ (\textsc{cut-ce})$$

Figure 2.14: Rules for the definition of cycleElimRel (simplified).

successors are not loop heads themselves, one must apply the following rules in order: (1) rule HAVOC-CE for each introduced `havoc` command, (2) rule PREINV-CE for the precondition (reflecting that the `assert` in the loop head is transformed into an `assume` command), (3) repeated application of rule ID-CE. Given the parameters to cycleElimRel it is always clear which rule to apply next and thus automating proofs for cycleElimRel judgements is straightforward.

Our approach for proving local block lemmas via cycleElimRel relies on Boogie transforming blocks in a certain way. For example, our approach relies on the inserted `assert` commands using the invariant precisely as it appears in the source blocks, and leaving blocks that are neither loop heads nor predecessors to loop heads unchanged. One could instead devise a more semantic approach that does not need to rely on such aspects. However, then automation becomes more challenging. Since we do not expect large changes in cycle elimination transformation, we decided to pick an approach that is easy to automate and also easy to adapt to smaller changes.

2.4.5. Global Block Theorems

The second step of our general validation approach is to lift the results from the local block lemmas to global block theorems, which concern executions through the *entire* source and target CFGs starting from some source and its corresponding target block. In essence, this lifted result implies that if there are no failing executions starting from some target block (possibly continuing through the CFG via successors), then there are no failing executions starting from the source block.

The major challenge in the proofs for global block theorems is reasoning about looping executions in the source CFG for three reasons. First, each iteration of the loop (of which there may be infinitely many) must possibly be justified by a *different execution* in the target CFG: these target executions start from the block corresponding to the source loop entry and end in a block B'_t corresponding to the end of an iteration in the target entry (without every continuing beyond B'_t). Second, we need to include sufficient properties in the global block theorems such that each block pair can be considered separately, and each global block theorem can be proved using the local block theorem of the same block pair and the global block theorems of the successors. This requires reflecting properties about the corresponding loop heads and the modified loop variables X_H identified by Boogie. An alternative to this modular approach would be to prove a single global block theorem for the entire loop directly. This would be cumbersome, since a loop consists of many blocks, may contain nested loops, and may have multiple back edges. Third, as discussed in Subsection 2.4.2, our proof needs to check that each identified loop head is the unique entry point of the corresponding loop l, and thus dominates all of the blocks in l.

In our approach, we deal with the first two points as follows. We reason about the possibly unbounded number of loop iterations by proving global block theorems involving a loop head by induction on the number of steps remaining in the execution in question.[18] To allow the corresponding induction hypothesis to be applicable at the end of the loop iteration, the induction generalises over the states in which executions from the loop head starts: any state is considered that differs on at most the variables X_H compared to the corresponding target state. To allow a modular approach that considers block pairs separately, our global block theorem for a block B within a loop then carries an additional assumption reflecting the induction hypothesis for each loop that contains B. Proving a global block theorem for the origin of a back edge is taken care of by applying the corresponding induction hypothesis for the case when the execution continues via the loop head. To prove a global block theorem within a loop, one must ensure that no variables outside of X_H are modified by the block, otherwise the induction hypothesis is not applicable at the end of the block.

18: This may seem insufficient since executions can be infinite, but importantly a *failing* execution is always finite, and our theorems need only eliminate the chance of failing executions.

With this setup, it is possible to prove all the global block theorems without inducing any circularities in the reasoning. Moreover, it turns out that our proof strategy also handles the third challenge: the proof strategy implicitly checks that the loop head dominates all the blocks in the corresponding loop. We will expand on these two points (*i.e.* proving the global block theorems without inducing circularities and implicit domination checks) in Subsection 2.4.6. First, let us take a closer look at the formal statement of our global block theorems:

Theorem 2.4.2 (Cycle elimination global block theorem) *For any block B, let $\mathsf{LH}(B)$ be the set of loop heads of the loops that B is contained in (excluding B if B is itself a loop head).*
Let Q_{post} be the postcondition of the procedure. Let G_S be the source CFG

and let G_T be the target CFG. Let B_S be a source block and let B_T be the corresponding target block. If B_S is a loop head, let X_H be as defined in cycle elimination step 1 (and empty otherwise) and let A_{pre} be its loop invariant (and **`true`** *otherwise).*
Let Λ be the variable context for the source and target procedures. Then, for any type interpretation $\mathcal{T}$, any well-formed function interpretation $\mathcal{F}$ w.r.t. $\mathcal{T}$, any states σ_1 and σ_2, any CFG configuration (m'_1, s'_1), and any number of execution steps i, if:

1. $(\mathcal{T}, \Lambda, \mathcal{F}), G_S \vdash (\mathsf{inl}(B_S), \mathsf{N}(\sigma_1)) \rightarrow^{i}_{CFG} (m'_1, s'_1)$
2. $\forall m'_2, s'_2.\ (\mathcal{T}, \Lambda, \mathcal{F}), G_T \vdash (\mathsf{inl}(B_T), \mathsf{N}(\sigma_2)) \rightarrow^{*}_{CFG} (m'_2, s'_2) \implies s'_2 \neq \mathsf{F}$
3. $\forall B_H \in \mathsf{LH}(B_S).\ \mathsf{loopIH}(\mathcal{T}, \Lambda, \mathcal{F}, G_S, B_H, \sigma_1, i, (m'_1, s'_1))$
4. *A_{pre} is satisfied in σ_1, and σ_2 matches σ_1 on all variables w.r.t. variable context Λ except for those in X_H*
5. *σ_1 and σ_2 are well-typed w.r.t. the variable context Λ where the types are interpreted wrt $\mathcal{T}$*

then*: $\mathsf{validConfig}(\mathcal{T}, \Lambda, \mathcal{F}, Q_{post}, (m'_1, s'_1))$ holds*

where validConfig captures the *validity of a CFG configuration* as defined by:

Definition 2.4.1 (Validity of a CFG configuration)

$$\mathsf{validConfig}(\mathcal{T}, \Lambda, \mathcal{F}, Q_{post}, (m', s')) \triangleq$$
$$s' \neq \mathsf{F} \wedge$$
$$\left(\begin{array}{l} m' = \mathsf{inr}(()) \Rightarrow \\ \quad \forall \sigma.\ s' = \mathsf{N}(\sigma) \Rightarrow (\mathcal{T}, \Lambda, \mathcal{F}), \emptyset \vdash \langle Q_{post}, \sigma \rangle \Downarrow \mathsf{BoolVal}(\mathit{true}) \end{array} \right)$$

The first assumption of the global block theorem expresses that a source execution exists starting from the source block that executes precisely i blocks. The conclusion states that the configuration reached by this execution must be valid. That is, the resulting outcome is non-failing, and if it is a normal outcome at the end of an exit block (*i.e.* a block without any successors), then the postcondition of the procedure holds. This conclusion is in contrast to the second assumption, which expresses that the corresponding target executions do not fail but does not say anything about the postcondition. The reason for this mismatch is due to Boogie inserting an **`assert`** command of the postcondition in exit blocks of the *target* CFG (the *source* CFG does not have these **`assert`** commands), as we will discuss in Subsection 2.4.8. Thus, just knowing that the target executions cannot fail is sufficient. The final two assumptions mimic assumptions used for the corresponding local block lemma.

The most interesting assumption is the third one reflecting the induction hypothesis for each loop in which the block is contained via loopIH. The definition of loopIH is given by:

Definition 2.4.2 (loopIH) *Let B_H be a loop head where X_H is as defined in cycle elimination step 1 and let A_{pre} be its loop invariant.*
Then, $\mathsf{loopIH}(\mathcal{T}, \Lambda, \mathcal{F}, G_S, B_H, \sigma, i, (m', s'))$ holds iff for all i' where $i' \leq i$ and all states σ' the following holds: if

1. $(\mathcal{T}, \Lambda, \mathcal{F}), G_S \vdash (\mathsf{inl}(B_H), \mathsf{N}(\sigma')) \rightarrow^{i'}_{\mathsf{CFG}} (m', s')$
2. A_{pre} *is satisfied in* σ'*, and* σ' *differs from* σ *only on variables in* X_H *and variables not defined in* Λ
3. σ *and* σ' *are well-typed w.r.t. the variable context* Λ *where the types are interpreted wrt* $\mathcal{T}$

then validConfig$(\mathcal{T}, \Lambda, \mathcal{F}, Q_{post}, (m', s'))$ *holds.*

loopIH for loop head B_H is very similar to the global block theorem for B_H. Apart from the source state σ and the source execution steps i potentially being different, there are the following three differences in loopIH compared to the global block theorem:

1. All numbers of execution steps i' that are most i are considered in the source (reflecting a *strong* induction) such that loopIH can be applied after any number of blocks executed in the loop.
2. There is no assumption stating that the target executions do not fail (assumption 2 in Theorem 2.4.2).
3. There is no assumption for each of the loops that B_H is contained in (assumption 3 in Theorem 2.4.2)

In point 1, the number of execution steps i' need not be *strictly* smaller than i because of the following reason. When doing an induction proof for the global block theorem of loop head B_H on the number of steps j of the source execution starting from B_H, we get a resulting induction hypothesis that is applicable for all executions from B_H that take *strictly* less than j steps. Since loopIH is used only for global block theorems of blocks within the corresponding loop *other than* the corresponding loop head B_H, the number of execution steps i considered in loopIH is always strictly smaller than j (since at least B_H must have been executed to reach blocks within the loop). Thus, the induction hypothesis is applicable for all number of execution steps that are at most i (*i.e.* including i).

The reason for points 2 and 3 is the following. When doing an induction proof for the global block theorem for loop head B_H, one may assume the following propositions: the resulting induction hypothesis and the premises in this global block theorem. These propositions are sufficient to directly justify the assumptions specified by points 2 and 3. So, these assumptions need not be included in loopIH. As a result, there is less work required when applying loopIH to justify executions that go back to a loop head via a back edge.

For point 2, for a given loop execution in the source CFG, all corresponding executions in the target CFG justifying each loop iteration start in the *same* state from the loop head (state σ_2 in Theorem 2.4.2). Thus, the required non-failure assumption on the target CFG is identical to the non-failure assumption obtained from the global block theorem of the loop head B_H.

For point 3, the assumption of the global block theorem of B_H already states that loopIH holds for the relevant loops where the source state parameter σ in loopIH is given by a fixed source state σ_1. To justify point 3, we need to essentially argue that loopIH also holds for any instantiation of the source state parameter that differs from σ_1 only on variables X_H. This is the case, because loopIH considers *all* executions starting from any state that differs from the source state parameter only on variables X_H.

2.4.6. Generating Proofs for Global Block Theorems

As we discuss in Subsection 2.2.2 on page 16, in our high-level approach, typically to prove the global block theorem for source block B, we use the local block lemma of B (to reason about the execution through B itself) and the global block theorems of the successors of B (to reason about executions continuing after B). This strategy leads to dependencies between global block theorems. We need to make sure that these dependencies do not induce circular reasoning, which is not obvious in the case of cycle elimination as the source CFG may have cycles. The way we avoid circularities is by proving the global block theorem *gbt* of the origin of a back edge e using the induction hypothesis corresponding to the target of e (which is a premise of *gbt*), instead of using the global block theorem of the target of e. This way the global block theorem of the target of e (*i.e.* a loop head) depends ultimately on *gbt* but *not* vice versa.

Isabelle does not allow circular reasoning by forcing one to prove Isabelle lemmas in a given order, where the proof of a lemma may use only lemmas that have been previously proved. So, we need to choose an order to explicitly show that our approach indeed does not result in circular dependencies. With our strategy, we are able to do so by automatically generating proofs for global block theorems in *reverse topological order* of the target CFG. Such an order exists, since the target CFG is acyclic. This order guarantees that the proof for the theorem of source block B can rely on the global block theorems of the successors of B excluding those successors reached via back edges (those can be justified via the induction hypothesis assumptions). This enables proving each global block theorem via the corresponding local block lemma and the global block theorems of the successors. For example, in our running example with a single loop shown in Figure 2.11 on page 38, a possible order is given by $B_6, B_5, B_3, B_4, B_2, B_1, B_0$. The blocks B_2, B_3, B_4, B_5 within the loop each have an induction hypothesis for the loop. The proof for the theorem of B_5 uses the induction hypothesis to justify executions continuing via the back edge to the loop head B_1. The proofs for the theorems of B_3 and B_4 need the induction hypothesis in order to use the theorem of B_5. The proof for the theorem of B_2 needs the induction hypothesis in order to use the theorem for B_3 and B_4. The proof for the loop head B_1 is done via an induction, which provides the induction hypothesis needed to use the theorem of B_2.

Our proof strategy works only if every path to a block B_l in a loop l first goes through the corresponding loop head B_H identified by Boogie (*i.e.* B_H dominates B_l). As discussed, this property holds only if the source CFG is reducible. If there were a path p that entered the loop via B_l without first going through the identified loop head, then our proof strategy would fail. The reason is that one of the premises of the global block theorem gbt_l of B_l is the induction hypothesis for B_H. As a result, the global block theorem of the predecessor B_{pre} of B_l in p would not be provable with our approach, since in this proof there would be no way of justifying the induction hypothesis premise of gbt_l (because B_{pre} is not in the loop, the global block theorem of B_{pre} would not have such a premise).

To concretely see why our proof strategy would fail as expected in cases where an identified loop head does not dominate all blocks in the corresponding loop, recall the *unsound* transformation shown in Figure 2.13 on page 40, where the loop head B_2 *does not* dominate the origin B_4 of a back edge. Here, since B_4 is a block in the loop, one of the premises in the corresponding global block theorem gbt_4 is the induction hypothesis for the loop. As a result, we would not be able to prove the global block theorem gbt_0 for B_0 (a predecessor of B_4 outside of the loop). To do so, we would have to use the global block theorem for B_4, which we cannot, since in the proof of gbt_0 there is no way to establish the induction hypothesis premise in gbt_4.

In other words, if our proof succeeds, then our proof implicitly shows the necessary requirement that loop heads (as identified by Boogie) dominate all the blocks in the corresponding loop *without us formalising any notion of domination, CFG reducibility, or any other advanced graph-theoretic concept*. This shows a major benefit of our validation approach over a once-and-for-all proof of Boogie itself: our proofs indirectly check that the identification of loop heads and back edges guarantees the necessary *semantic properties* without being concerned with *how* Boogie's implementation computes this information.

Our approach applies equally to nested loops and more-generally to reducible CFG structures. For example, the global block theorem for block B_3 that is in a nested loop in Figure 2.12 on page 39 may assume induction hypotheses for both the inner and outer loop. In nested loops, the requirement that no more than the havocked variables X_H are modified in the outer loop is easily handled by showing that variables modified in an inner loop are a subset of the havocked variables defined in the corresponding outer loops.

To express and prove the global block theorems, our instrumentation directly extracts the loop head and back edge identification from Boogie. Thus, this identification need not be recomputed. From the loop heads and back edges, it is straightforward to compute the loops in which a block is contained, which is needed to express the assumption for the induction hypotheses in our global block theorems. As for all parts of our per-verifier-run generated certificates, our global block lemmas and corresponding proofs are automatically generated and then automatically checked in Isabelle per Boogie procedure.

2.4.7. Proving Soundness of this Transformation Subsequence

To prove the soundness of our chosen transformation subsequence involving cycle elimination, we need to show that if the target procedure is correct, then the source procedure is correct. To do so, we use the global block theorem relating the entry blocks in the source and target CFGs representing the procedure bodies.

As discussed in Subsection 2.3.4, a procedure is correct, if the procedure body has no failing executions for any well-formed type and function interpretation, and any well-typed initial state in which the axioms are satisfied.[19] Since the transformation subsequence involving cycle

19: We ignore pre- and postconditions here for the sake of presentation; they are handled in our generated certificates.

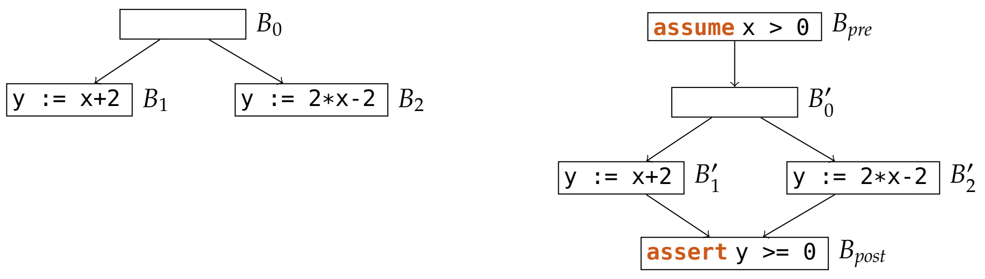


Figure 2.15: An example showing the insertion of precondition `x > 0` and postcondition `y >= 0` applied to the left CFG, where the resulting CFG is shown on the right.

elimination changes only the procedure bodies (*e.g.* the variables, uninterpreted functions, uninterpreted types, and axioms are identical), the well-formed type and functions interpretations and relevant initial states are the same for both the source and target procedure. Thus, to prove the desired result, we may consider an arbitrary well-formed type and function interpretation, and an arbitrary initial state in which the axioms are satisfied, and then prove that if the target procedure body has no failing executions w.r.t. these parameters, then neither does the source procedure body w.r.t. the same parameters. This follows directly from the global block theorem relating the entry blocks.

2.4.8. Pre- and Postcondition Insertion, Empty Block Insertion

So far in this section, we have discussed only the cycle elimination transformation. However, as shown in Figure 2.1 on page 17, our generated certificate for the corresponding transformation subsequence also includes two simple transformations: the insertion of pre- and postconditions, and the insertion of empty blocks. One consequence of both of these transformations is that not every block in the target CFG has a corresponding block in the source CFG.

Figure 2.15 shows an example of pre- and postcondition insertion where the precondition is `x > 0` and the postcondition is `y >= 0`. Here, two new blocks are added: a new entry block B_{pre} where the precondition is assumed and a new exit block B_{post} where the postcondition is asserted. These two blocks do not have corresponding blocks in the source. We handle the pre- and postcondition insertion transformation in our proof generation as follows. For each source block and its corresponding target block, we prove a a global block theorem as discussed for cycle elimination. In Figure 2.15, this means we prove theorems for the pairs (B_0, B'_0), (B'_1, B_1), (B_2, B'_2). To prove the global block theorems for the source exit blocks (*i.e.* pairs (B_1, B'_1) and (B_2, B'_2) in the figure), we prove that executions reaching the end of the corresponding target blocks can continue to the new unified target exit block (*i.e.* B_{post} in the figure) and thus the postcondition must hold since those target executions cannot fail.[20] Finally, we prove a global block theorem relating the source entry block (*i.e.* B_0 in the figure) with the new target entry block that has the inserted precondition (*i.e.* B_{pre}). Here, we need to include an additional assumption in the global block theorem stating that the precondition holds in the initial state in order to justify the `assume` command in the

20: Here, we must again use the type soundness of expressions to show that the postcondition reduces to a Boolean.

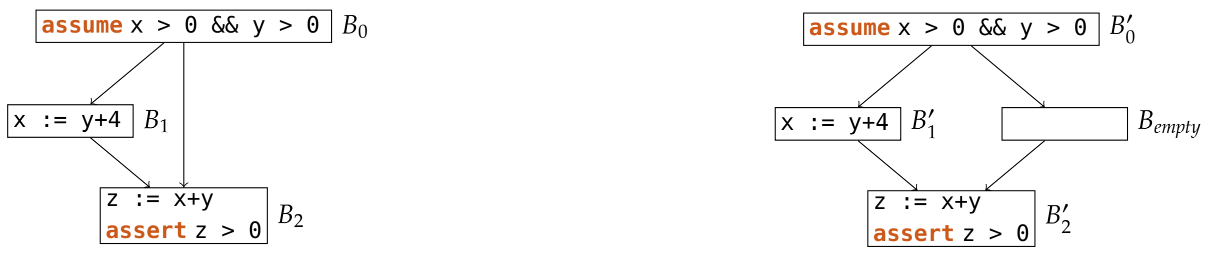


Figure 2.16: An example showing the insertion of empty blocks applied to the left CFG, where the resulting CFG is shown on the right.

target block. The resulting global block theorem can be used to prove the soundness of the transformation, since the correctness of a Boogie procedure considers only those executions that satisfy the precondition.

The empty block insertion transformation inserts empty blocks to ensure that every block B with more than one predecessor (*i.e.* a *join block*) has only predecessors with exactly one successor (namely B). This property is important for the soundness of assignment elimination, which will become clear in Subsection 2.5.6.

The left CFG in Figure 2.16 shows an example where the mentioned property does not hold. In particular, B_2 has two predecessors and one of the predecessors (B_0) has two successors. As a result, Boogie inserts an empty block B_{empty} between B_0 and B_2 to obtain the CFG on the right, which establishes the property.

Handling these empty blocks in the proof generation is straightforward. For global block theorems where the target block has an inserted empty successor, we prove that the execution in the target CFG can continue to the original successor. For example, in Figure 2.16 for the global block theorem relating B_0 and B'_0, we prove that target executions continuing from B'_0 to B_{empty} further continue to B'_2, and then we can use the global block theorem relating B_2 and B'_2.

2.5. Assignment Elimination

In this section, we describe our certificate generation approach for the assignment elimination transformation. As part of assignment elimination, the Boogie verifier performs constant propagation and desugars old expressions. For the sake of presentation, we will focus on assignment elimination *without* the other two modifications first, and at the end discuss how to handle assignment elimination *with* them. Unlike the case of cycle elimination, assignment elimination makes no changes to the CFG structure, but makes substantial changes to the program states (via SSA-like renamings), substantially increases nondeterminism, and employs `assume` commands to re-tame the sets of possible executions.

2.5.1. Assignment Elimination Overview

The main goal of assignment elimination is that ultimately a more efficient VC can be generated [21–23]. In the Boogie verifier, this is

[21]: Flanagan et al. (2001), *Avoiding exponential explosion: generating compact verification conditions*
[22]: Leino (2005), *Efficient weakest preconditions*
[23]: Barnett et al. (2005), *Weakest-precondition of unstructured programs*

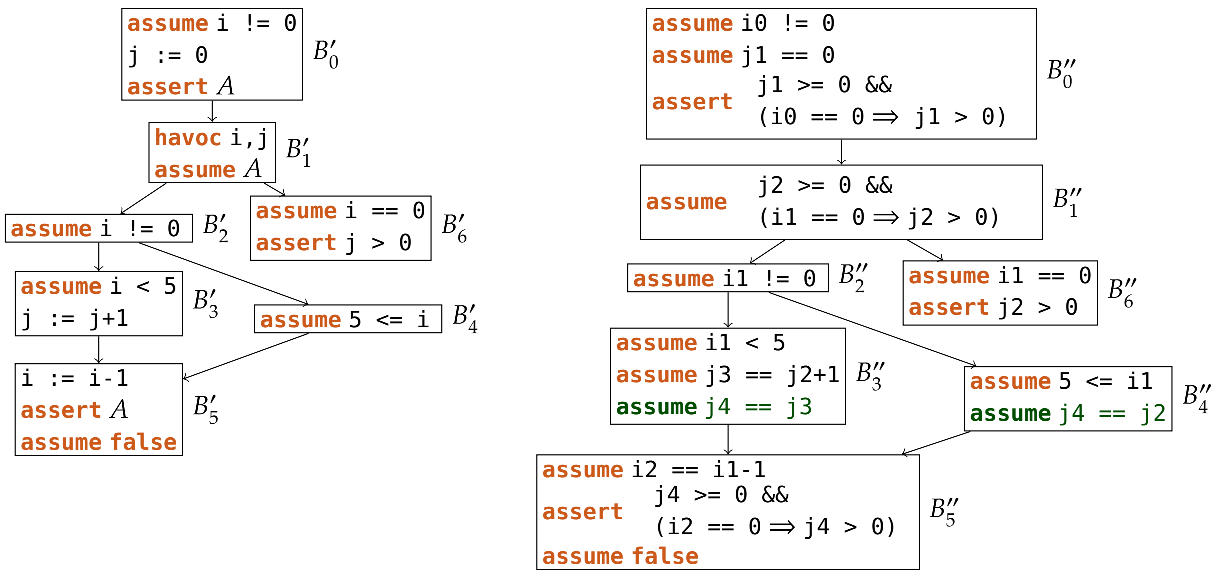


Figure 2.17: Assignment elimination applied to the running example where A is given by `j >= 0 && (i == 0 ⇒ j > 0)`. The CFG after the previous transformations is shown on the left and the result of assignment elimination is shown on the right. In practice, Boogie also applies constant propagation for the assignment `j := 0` as part of assignment elimination, which we ignore in this figure for the sake of presentation. The final (green) commands in B''_3 and B''_4 are the synchronisation commands. On entry of source block B'_2, the current versions of `i` and `j` are `i1` and `j2`, respectively.

implemented as a single transformation that can be thought of as two independent steps. Firstly, the source CFG is transformed into *static single assignment* (SSA) form, introducing *versions* (fresh variables) for each original program variable such that each version is assigned to at most once in any program execution. In a second step, variable assignments are *completely eliminated*: each assignment command $x := e$ is replaced by **assume** $x == e$. Moreover, **havoc** commands are simply removed; their effect is implicit in the fact that a new variable version is used (via the SSA step) *after* such a command.

Figure 2.17 shows the effect of this transformation on our running example. The *synchronisation commands* (highlighted in green) inserted just before the join block (here, B''_5) introduce a consistent variable version (here, `j4`) for use in the join block. It is convenient to speak of target variables in terms of their source program counterparts: we say *e.g.* that `j` *has version* 4 on entry to block B'_5.

Compared to executions through the source program, the space of executions through the target program is much wider, because for each source variable there is a set of versioned target variables. In particular, the values of each versioned variable in the target program are initially unconstrained, meaning executions exist for all of their combinations. The target program constrains a versioned variable via an **assume** command at the point where the corresponding (unique) update occurs in the source program. These **assume** commands ensure that the versioned variable captures precisely the corresponding source variable *after* the corresponding update. Some of the target executions do not survive these **assume** commands and go to magic. Importantly, however, not *all* executions go to magic; enough are preserved to simulate the executions of the original program. This is because each **assume** command constrains the value of exactly one variable version, and the same version is never constrained more than once (guaranteed by the SSA step). Thus, for each full source execution through the source CFG, there must exist one target execution corresponding to the source execution, namely the one for which the variable versions hold precisely the values that are assigned to the variables in the source execution at the corresponding (unique) points. Capturing this delicate argument formally is the main challenge in certifying assignment elimination.

As an example, consider the variable `j4` in the target CFG in Figure 2.17. The variable `j4` remains unconstrained in the target CFG from the entry block until blocks B''_3 and B''_4. Executions through B''_3 (or B''_4) where `j4` does not correspond to the constraint of the **`assume`** command in B''_3 (or B''_4) will go to magic and not reach B''_5. However, since `j4` is guaranteed to be previously unconstrained, there is at least one simulating execution where `j4` has the value for which the constraint is fulfilled.

2.5.2. Local Block Lemmas

To validate assignment elimination, it is sufficient to prove the following property P: each source execution through the source CFG is simulated by a *single* target execution through the target CFG, made precise by constructing a relation between the states in these executions. A natural approach for the local block lemma might therefore be to show that each source execution through a single source block is simulated by a *single* target execution through the corresponding target block.[21] Then, the idea would be to compose the local block lemmas (via the global block theorems) to construct the simulating target execution for any execution through the source CFG starting from the entry block. Such *forward simulation* arguments to prove the property P are standard. However, this approach for proving local block lemmas does not work for Boogie's assignment elimination transformation.

21: Such a local block lemma would be similar to the one for cycle elimination (see Subsection 2.4.3).

The reason such a forward simulation approach does not work here is that given an initial target state, the target CFG does not modify any variables and thus leaves states unchanged. As a result, choosing a *single* simulating target execution through a block B requires choosing a state for the target execution, which directly defines *every* value for each of the variable versions (including future versions that are relevant *after* the execution of B). Thus, in a local block lemma, picking the correct target execution would require knowledge of all variable assignments in the source CFG that are *going* to happen, which is not possible due to nondeterminism. So, to nevertheless allow a block-modular certification strategy, we must tackle this challenge of expressing a local block lemma for assignment elimination via another approach.

To illustrate this challenge on an example, consider relating the block B'_0 and B''_0 in Figure 2.17. Given a source execution through just the block B'_0 in a state σ, we would have to find a corresponding target execution for B''_0. However, it is impossible to, for example, identify a single correct value for `j4` in the target state that will work for every possible continuation of the source execution after B'_0. In particular, if the **`havoc`** commands in B'_1 will choose a value smaller than 5 for `i` and choose value 1 for `j`, then the value for `j4` should be 2 (since only B'_3 will reach a normal outcome).

But if the **havoc** commands in B_1' instead will choose a value greater than 5 for `i` and choose value 1 for `j`, then the value for `j4` should be 1 (since only B_4' will reach a normal outcome). Since these choices depend on nondeterministic choices in the future that are not determined by the current states (*e.g.* source state σ), it is not possible to make a single choice that will work in all cases.

To handle this challenge, we relate each source state with a *set* T of corresponding target states (instead of a *single* target state). This allows us to choose a set of target executions for the local block lemma such that for each possible future continuation of the source execution at least one of the target executions will be the correct one simulating the source execution.

To specify the relationship between source and target states, we define variable relations $\mathcal{V}_R$ at each point in a execution, making explicit the mappings used in the SSA step between source program variables and their corresponding versions. For example, on entry to block B_2' in the source version of our running example (correspondingly B_2'' in the target), the $\mathcal{V}_R$ relation relates `i` to `i1` and `j` to `j2`. All target states $t \in T$ must precisely agree with the source state s w.r.t. $\mathcal{V}_R$ (*e.g.*, $s(\texttt{i}) = t(\texttt{i1})$, $s(\texttt{j}) = t(\texttt{j2})$). On the other hand, our sets of states T are required to be completely unconstrained (besides typing) for *future* variable versions. For example, for every $t \in T$ at the same point in our example, there must be states in T assigning each possible value (of the same type) to `i2` (and otherwise agreeing with t).

More precisely, for a set of variables X, we say that a set of states T *constrains at most X w.r.t. variable context Λ* if, for every $t \in T$, for every $z \notin X$ s.t. z is recorded in Λ, and for every value v of z's type w.r.t. Λ, we have $t[z \mapsto v] \in T$. In other words, the set T is closed under arbitrary changes to values of all variables in Λ but *not* in X. We construct our sets T such that they constrain at most *current and past versions* of program variables. It is this fact that enables us to handle subsequent **assume** commands in the target program and, in particular, to show that the set of possible executions in the target program never becomes empty while there are possible executions in the source program. For example, when relating the source command `j := j+1` in B_3' with the target command **assume** `j3 == j2 + 1` in block B_3'', we use the fact that our set of states does not constrain `j3` to prove that, although many executions go to magic at this point, for a non-empty set of states $T' \subseteq T$ (those in which `j3` has the "right" value equal to `j2 + 1`), the execution continues in the target.

We now make these notions more precise by showing our local block lemmas for assignment elimination:[22]

22: We omit some details regarding well-typedness, handled fully in our Isabelle formalisation.

Theorem 2.5.1 (Assignment elimination local block lemma) *Let cs_S be the commands of a source block B whose corresponding target block has commands cs_T. Let Λ_1 and Λ_2 be the variable contexts for the source and target procedures, respectively. Let $\mathcal{V}_R$ and $\mathcal{V}_R'$ be the variable relations at the beginning and end of B, respectively. Let Y be the variable versions corresponding to the target variables of assignment and* **havoc** *commands in cs_S. Then, for any type interpretation $\mathcal{T}$, any function interpretation $\mathcal{F}$, any set of variable versions X, any state σ, any outcome s', and any non-empty* *set T of states such that σ agrees with T according to $\mathcal{V}_R$ and T constrains at most X w.r.t. Λ_2, it must hold that: if*

1. $(\mathcal{T}, \Lambda_1, \mathcal{F}) \vdash \langle cs_S, \mathsf{N}(\sigma)\rangle\ [\rightarrow]\ s' \wedge s' \neq \mathsf{M}$
2. $X \cap Y = \emptyset$

then there exists a non-empty set of states $T' \subseteq T$ s.t. T' constrains at most $X \uplus Y$ w.r.t. Λ_2 and for each $\sigma'_t \in T'$, there exists an outcome s''_t s.t.

1. $(\mathcal{T}, \Lambda_2, \mathcal{F}) \vdash \langle cs_T, \mathsf{N}(\sigma'_t)\rangle\ [\rightarrow]\ s''_t \wedge (s' = \mathsf{F} \Longrightarrow s''_t = \mathsf{F})$
2. *If s' is a normal outcome, then the state of s' and σ'_t are related w.r.t. $\mathcal{V}'_R$, and $s''_t = \mathsf{N}(\sigma'_t)$ holds.*

This lemma captures our generalised notion of forward simulation appropriately. The universally quantified set of variable versions X represents the previous and current variable versions on entry of the blocks.[23] Given a set of *non-empty* target states T related to the source state and constrained at most by X, the lemma ensures that there is a non-empty subset T' of T such that (1) each target execution from any state $\sigma'_t \in T'$ simulates the provided source execution through the block, and (2) T' is constrained at most by X and the newly constrained variables Y, ensuring that future variable versions remain unconstrained.

23: X is universally quantified instead of being fixed to a concrete set in order to give the flexibility of using this lemma for different sets.

The two conclusions in the lemma make the simulation precise. The first conclusion in the lemma expresses that the target does not get stuck and that failures are preserved, while the second shows that if the source execution executes normally then the resulting states are related. Note that premise 2 ($X \cap Y = \emptyset$) is essential in the proof to guarantee that the `assume` commands introduced by assignment elimination in the target block do not eliminate the chance to simulate source executions; the condition expresses that the variable versions newly constrained do not intersect with those previously constrained. To prove such a lemma for a concrete target block with commands cs_T, we must also check that the same version is not constrained twice in cs_T. Note that in these proofs and also in the proofs for our global block theorems, we need not deal with any explicit notion of Boogie's SSA computation and also need not explicitly formalise the precise property guaranteed by SSA. Instead, we prove simple disjointness checks, which are guaranteed to hold if Boogie's SSA computation is correct, and which together show the necessary conditions for assignment elimination to be sound.

2.5.3. Generating Proofs for Local Block Lemmas

To automatically generate proofs for the local block lemmas for assignment elimination such that Isabelle can automatically check them, we use a similar approach as for cycle elimination. We define a relation on two lists of basic commands, which captures the possible syntactic relationships between a source CFG block and target CFG block. Then, we prove once and for all that if two blocks are in this relation, then under certain assumptions A, the corresponding local block lemma holds. Thus, generating a proof for a concrete local block lemma boils down to generating proofs showing that (1) the corresponding source and target blocks are in this relation, and (2) the assumptions A hold.

$$\frac{\mathsf{expRel}(\mathcal{V}_R, e, e') \quad \mathsf{assignElimRel}(\mathcal{V}_R[x \mapsto x_i], W, cs_S, cs_T)}{\mathsf{assignElimRel}(\mathcal{V}_R, (x, x_i) :: W, x := e :: cs_S, \texttt{assume}\ x_i == e' :: cs_T)}\ (\textsc{assign-ae})$$

$$\frac{\mathsf{assignElimRel}(\mathcal{V}_R[x \mapsto x_i], W, cs_S, cs_T)}{\mathsf{assignElimRel}(\mathcal{V}_R, (x, x_i) :: W, \texttt{havoc}\ x :: cs_S, cs_T)}\ (\textsc{havoc-ae})$$

$$\frac{\mathcal{V}_R(x) = x_i \quad \mathsf{assignElimRel}(\mathcal{V}_R[x \mapsto x_j], W, [], cs_T)}{\mathsf{assignElimRel}(\mathcal{V}_R, (x, x_j) :: W, [], \texttt{assume}\ x_j == x_i :: cs_T)}\ (\textsc{sync-ca})$$

$$\frac{\mathsf{expRel}(\mathcal{V}_R, e, e') \quad \mathsf{assignElimRel}(\mathcal{V}_R, W, cs_S, cs_T)}{\mathsf{assignElimRel}(\mathcal{V}_R, W, \texttt{assert}\ e :: cs_S, \texttt{assert}\ e' :: cs_T)}\ (\textsc{assert-ae})$$

$$\frac{\mathsf{expRel}(\mathcal{V}_R, e, e') \quad \mathsf{assignElimRel}(\mathcal{V}_R, W, cs_S, cs_T)}{\mathsf{assignElimRel}(\mathcal{V}_R, W, \texttt{assume}\ e :: cs_S, \texttt{assume}\ e' :: cs_T)}\ (\textsc{assume-ae})$$

$$\frac{}{\mathsf{assignElimRel}(\mathcal{V}_R, [], [], [])}\ (\textsc{nil-ae})$$

Figure 2.18: Rules for the definition of assignElimRel (simplified).

We define a relation between source block commands cs_S and target block commands cs_T w.r.t. the variable relation $\mathcal{V}_R$ at the beginning of the blocks of the following form (we will show the definition shortly):

$$\mathsf{assignElimRel}(\mathcal{V}_R, W, cs_S, cs_T) \tag{2.2}$$

The parameter W is a list of tuples representing the variable updates in cs_S, where the first element of the tuple is the updated source variable (*i.e.* the left-hand side of an assignment or the variable in a `havoc` command) and the second element of the tuple is the corresponding target version tracking the source variable after the update. The order of the list must correctly reflect the order in which the updates are performed in cs_S.

We prove once and for all that if the proposition 2.2 holds, then the corresponding local block lemma holds under the following assumptions:[24] (1) No target variable is constrained twice in cs_T, (2) The variable relation at the end of the block is given by the variable relation $\mathcal{V}_R$ at the beginning of the block adjusted by the updates specified by W. So, to prove a concrete local block lemma relating a source block with commands cs_S and a target block with commands cs_T, we need to do the following: (1) identify the variable relation $\mathcal{V}_R$ at the beginning of the blocks and the updates W for these blocks, and then (2) prove $\mathsf{assignElimRel}(\mathcal{V}_R, W, cs_S, cs_T)$. We are able to obtain $\mathcal{V}_R$ and W by instrumenting the existing Boogie verifier implementation, which keeps track of the variable relation explicitly and which contains code that performs the variable updates.

24: We ignore typing-related assumptions for the sake of presentation; they are fully handled in our formalisation.

The relation assignElimRel is defined inductively. Figure 2.18 shows a simplified version of the rules. Most rules syntactically match a command in the source with a command in the target; other rules just match a command in the source or target, which then results in a change in the variable relation in the premise. Since it is always clear which rule to apply next, it is straightforward to write an Isabelle tactic that automatically proves that a source and a target block are in the relation.

The rules ASSIGN-AE and HAVOC-AE capture variable updates in the source program via assignments and `havoc` commands, respectively. For both rules, one of the premises checks the remaining commands with the updated variable relation. Note that in the case of HAVOC-AE, the parameter for the variable updates in assignElimRel is used to obtain the new target variable version tracking the source variable after the `havoc` command, while for ASSIGN-AE one could obtain the information by inspecting the target `assume` command.

The judgement $\mathsf{expRel}(\mathcal{V}_R, e, e')$, which occurs as a premise in multiple rules, ensures that e and e' evaluate to the same values in states related via $\mathcal{V}_R$. We define separate inductive rules for expRel (not shown here). The rules are straightforward: they make sure that the expressions have the same structure and that the variables are related via $\mathcal{V}_R$.

The rule SYNC-CA handles synchronisation commands in the target block (such as, for example, the final commands in B_3'' and B_4'' in Figure 2.17). Here, the parameter for the variable updates is used to obtain the corresponding source variable that is related to the synchronisation command.

2.5.4. Global Block Theorems

As for all transformation subsequences, we lift our local block lemmas to global block theorems certifying all executions *starting* from a particular block, and thus, ultimately, to entire CFGs. For assignment elimination, the main conceptual challenges are taken care of by the local block lemmas. As a result, the global block theorems are a direct generalisation of the local block lemmas and we prove the theorems in reverse topological order of the source and target CFGs. Formally, our global block theorems are expressed as follows:[25]

25: We omit some details regarding well-typedness, handled fully in our formalisation.

Theorem 2.5.2 (Assignment elimination global block theorem) *Let B_S be a source block in the source CFG G_S and let B_T be the corresponding target block in the target CFG G_T. Let Λ_1 and Λ_2 be the variable contexts for the source and target procedures, respectively. Let $\mathcal{V}_R$ be the variable relation at the beginning of B_S and B_T. Let Z be the set of variables that have not yet been constrained beginning from B_T in G_T. Then, for any type interpretation $\mathcal{T}$, any function interpretation $\mathcal{F}$, for any set of variable versions X, any state σ, any CFG configuration (m_1', s_1'), any non-empty set of states T such that σ agrees with T according to $\mathcal{V}_R$ and T constrains at most X w.r.t. Λ_2, it must hold that: if*

1. $(\mathcal{T}, \Lambda_1, \mathcal{F}), G_S \vdash (\mathit{inl}(B_S), \mathsf{N}(\sigma)) \rightarrow^*_{CFG} (m_1', s_1')$
2. $X \cap Z = \emptyset$
3. $\forall \sigma_t \in T.\ \forall m_2', s_2'.$
   $(\mathcal{T}, \Lambda_2, \mathcal{F}), G_T \vdash (\mathit{inl}(B_T), \sigma_t) \rightarrow^*_{CFG} (m_2', s_2') \Longrightarrow s_2' \neq \mathsf{F}$

then $s_1' \neq \mathsf{F}$

Here, as in the local block lemma, X is an arbitrary set representing the current and past variable versions (we instantiate this set X with the concrete initial versions for the global block theorem relating the entry blocks; see Subsection 2.5.7). The set Z includes the set Y of new variables (from the corresponding local block lemma) that are newly constrained within the target block B_T, but also includes any variables newly constrained in any block reachable from B_T.

Note that the third premise in the global block theorem here (non-failure in the target) is analogous to the second premise in the global block theorem for cycle elimination (Theorem 2.4.2 on page 45). The only difference is that here non-failure in the target CFG is considered for *every* state $\sigma_t \in T$, while in the case of cycle elimination only a single state is considered. Also note that the global block theorem here does not say anything about the postcondition in contrast to the theorem for cycle elimination. The reason is that the postcondition is inserted into the CFG prior to assignment elimination (see Subsection 2.4.8) and thus the global block theorem need not consider the postcondition here.

2.5.5. Generating Proofs for Global Block Theorems

We automatically generate a proof for a global block theorem relating a source block B_S and a target block B_T via the corresponding local block theorem and the global block theorems relating the successors of B_S and B_T. To do so, we need to prove two properties on the unconstrained versions Z at the beginning of B_T. First, to use the local block theorem, we need to prove that the newly constrained variables Y within the execution of B_T are a subset of Z. Second, to use the global block theorem relating a successor B_{succ} of B_T, we need to prove that the unconstrained variable versions *at the beginning* of B_{succ} are a subset of the unconstrained versions *at the end* of B_T (the latter is given by $Z \setminus Y$).

Tracking and checking the subset of these concrete sets of unconstrained variable versions is simple, but gets expensive in Isabelle when the sets are large. In particular, representing these sets leads to large terms, which must be parsed and type-checked by Isabelle. Moreover, to check the subset between large sets, Isabelle needs to perform many rewrite steps. We circumvent this issue with our own *global versioning scheme* (as opposed to the versions used by Boogie, which are *independent* for different source variables): according to the CFG structure, we assign a unique *global* version number $\mathsf{ver}_{\mathcal{G}}(x)$ to each variable x in the target program such that, if x is first constrained in a target block B' and y is first constrained in another target block B'' reachable from B', then $\mathsf{ver}_{\mathcal{G}}(x) < \mathsf{ver}_{\mathcal{G}}(y)$. We can then represent the unconstrained variable versions efficiently via the minimum global version z that has not yet been constrained; z represents the set $\{x \mid x \geq z\}$. As a result, one does not have to make every single unconstrained variable version explicit.

Using this representation and the ordering property on the global version, we can encode our subset properties much more cheaply via an integer comparison. To check that $M \subseteq M'$, we just need to check if the minimum version in M is at least the minimum version in M'. Moreover, we can represent the set $Z \setminus Y$ in the second property (where Y is a finite set) via the maximum version in Y plus one if Y is non-empty (since we also check that $Y \subseteq Z$).[26] Since we represent variables as integers in the mechanisation, we directly use our global version *as* the variable name for the target program; there is no need for an extra lookup table when doing these subset checks. Note that (readability aside) it makes no difference which variable names are used in intermediate CFGs; we ultimately care only about validating the original CFG.

26: Note that $\{x \mid x \geq max(Y) + 1\}$ is potentially a strict subset of $Z \setminus Y$ (*i.e.* when there are versions smaller than $max(Y)$ in Z that are not in Y). This is fine, since the global version property ensures that any future reachable block will only use versions larger than $max(Y)$.

One question is whether one can always compute such a global versioning. If this were not the case, then our proof strategy with the more efficient set computations would be incomplete: we would not be able to generate a correct proof (and thus, Isabelle would not be able to check the proof successfully) even though the transformation itself may be sound. However, it turns out that for target programs generated by Boogie it is possible to compute such a global versioning.

The main challenge to compute the existence of such a global versioning is dealing with the synchronisation variables, which are constrained in the predecessors of join blocks to synchronise executions (*e.g.* `j4` in Figure 2.17). The reason is that these are the only variables not constrained exactly as part of a single introduced **`assume`** command in the target CFG and thus there are more ordering constraints that the global version must ensure for these variables. To handle this challenge, we compute a topological order on the blocks (starting from the entry block) where the predecessors P of a join block appear together consecutively in the order (this implies that all predecessors of blocks in P appear in the order before any block in P itself). We then iterate over the blocks in this order and provide fresh global versions to each of the constrained variables in the blocks incrementally (increasing the version by one, each time we assign a new version). Whenever we reach a consecutive sequence P of predecessors of a join block in the topological order, we assign versions to the synchronisation variables that are larger than those provided to ancestor blocks of P (*i.e.* a block from which some block in P is reachable), since we have already processed each ancestor block of P by that point.

One important reason why such a particular topological order exists and the corresponding computation to establish a correct global versioning is possible is because the empty block insertion (see Subsection 2.4.8) ensures that every predecessor of a join block has only one successor (namely the join block). In particular, this means that no predecessor of a join block B is reachable from any other predecessor of B, and thus the global versions need not deal with constraints between the predecessors. In the next subsection, we will discuss a concrete example where assignment elimination would become unsound if empty blocks were not inserted.

2.5.6. Two Important Properties

For assignment elimination to be sound in general, two important properties must hold on the source CFG, which are guaranteed by transformations prior to assignment elimination. The first property is that the source (and thus target) CFG is acyclic, which is guaranteed by cycle elimination. The second property is that the predecessor of every join block B has exactly one successor (namely B), which is guaranteed by

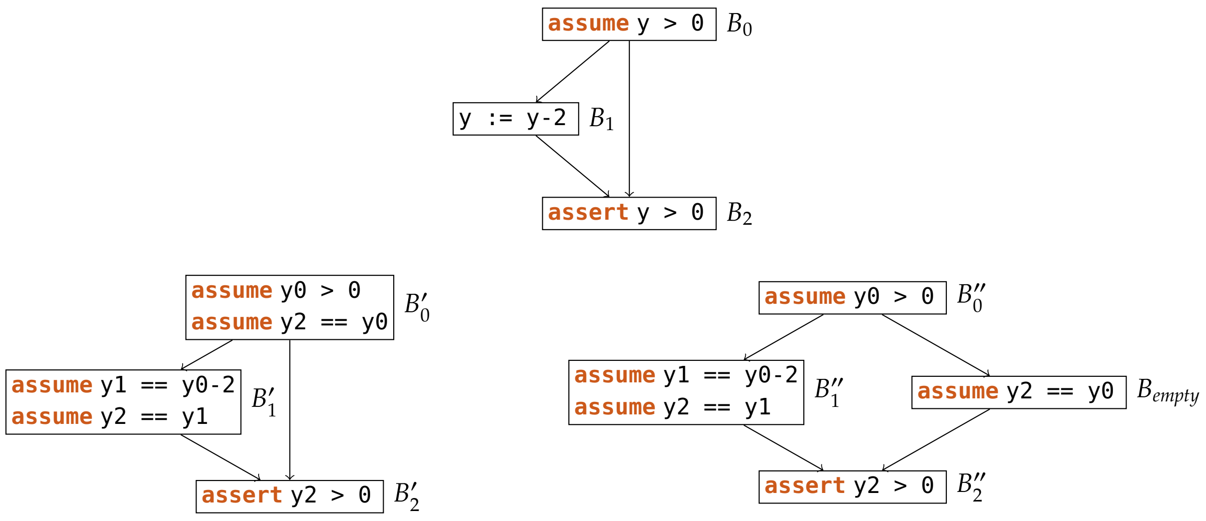


Figure 2.19: An example showing why Boogie inserts empty blocks before eliminating assignments.

empty block insertion. We make neither of these properties explicit. If the properties would not hold, then our proofs would fail. This shows another instance where a per-run validation approach is simpler than a once-and-for-all certification approach, which would likely have to somehow make the two properties explicit. This is similar to how our validation strategy for cycle elimination does not make any notion of domination or CFG reducibility explicit.

If the first property (the source CFG is acyclic) did not hold, then assignment elimination would be unsound in general because of the following: A source update in a loop would not be captured by a single target version v_t, since the source update in a block B could execute multiple times assigning different values in the same execution. As a result, one could not use a single `assume` command to model the update. In such a case, our proof strategy for global block theorems would not work for two high-level reasons. First, we require that the global block theorems of successors of a block B can be independently proved from the global block theorem of B. If the source CFG has cycles, our approach would lead to circular dependencies and thus our proof strategy would fail. (For cycle elimination, we devised an approach tailored to cycle elimination that avoided circular dependencies arising from cycles in the source CFG; such an approach would not work for assignment elimination.) Second, there is no global versioning that ensures the ordering property for cyclic CFGs in general, and thus one of the required properties on the versions that we check as part of the proof would fail.

The second property's purpose (the predecessor of every join block B has exactly one successor) is to ensure that variables are correctly synchronised at join blocks. If the property did not hold, then synchronising variables at join blocks by simply adding the corresponding `assume` commands in the predecessors would be unsound in general. Figure 2.19 illustrates this. Here, the source CFG at the top has a join block with a predecessor that has multiple successors. The result of assignment elimination applied to this source CFG directly is shown at the bottom left. This result does not correctly capture the source CFG, because the execution going from B'_0 to B'_1 constrains the synchronisation variable y2 twice. As a result, this execution always goes to magic when reaching the final command in B'_1, since y2 and y1 never match. Thus, this target CFG is correct (the executions going directly from B'_0 to B'_2 never fail), even though the source CFG is not, which means the transformation is unsound here. The transformation is sound if one first introduces an empty block between B_0 and B_2 (which Boogie does), which establishes the second property. This correct result is shown on the bottom right of Figure 2.19.

2.5.7. Proving Soundness of this Transformation Subsequence

To prove the soundness of assignment elimination, we need to show that if the target procedure is correct, then the source procedure is correct. To do so, we use the global block theorem relating the entry blocks in the source and target CFGs representing the relevant procedure bodies; we will refer to this theorem as the *global entry theorem*. This proof is more involved than for cycle elimination (see Subsection 2.4.7 on page 48) because (1) assignment elimination changes the declared variables (in addition to the procedure bodies); thus the variable contexts and states are different for the source and target, and (2) the global entry theorem quantifies over a *set* of target states that capture a source state instead of quantifying over a single target state.

To prove that the source procedure is correct (recall the formal definition shown in Definition 2.3.1 on page 32), we consider an arbitrary well-formed type interpretation $\mathcal{T}$, an arbitrary well-formed function interpretation $\mathcal{F}$, and an arbitrary well-typed state σ_s that satisfies the axioms (when restricted to the constants).[27] We then prove that the source procedure body has no failing executions starting from σ_s with respect to the source Boogie context $\Gamma_s = (\mathcal{T}, \Lambda_s, \mathcal{F})$, where Λ_s is the source procedure's variable context.

27: The definition quantifies over the global and local state separately. σ_s includes both of them.

To conclude the proof, we must instantiate the universally quantified variables in the global entry theorem such that the theorem's conclusion gives us the desired result, and must show the theorem's premises. The most interesting part is the instantiation of the set of target states T that must be related with the source state σ_s and for which we must show that there is no failing target execution from any state in T by using the correctness of the target procedure. We instantiate T with the set of target states that (1) are well-typed w.r.t. $\mathcal{T}$ and the target variable context Λ_t, and (2) agree with σ_s according to the initial variable relation $\mathcal{V}_R$ (*i.e.* the initial variable versions in the target states are constrained using the corresponding source variable values in σ_s). We show that there are no failing executions for an arbitrary target state $\sigma_t \in T$ by using the correctness of the target procedure instantiated with $\mathcal{T}$, $\mathcal{F}$, and σ_t. All premises of the correctness definition can be shown to hold, which concludes the result. In particular, it is straightforward to show that the axioms are satisfied in σ_t because in practice the names of the constants in the target have the same names as the source variables. Since our instantiation of T also makes sure that constants in σ_t and σ_s coincide (constants are a subset of the variables constrained by the initial variable relation), and since we know that the axioms are satisfied in σ_s, the axioms must also be satisfied in σ_t (the axioms are the same in the source and target).

Apart from this, one still needs to prove the remaining constraints of the global entry theorem. For instance, we instantiate the universally quantified variable X in the global entry theorem with the set of variable versions that are constrained before the target procedure body is executed (*i.e.* all initial variable versions corresponding to the source variables and procedure arguments). We must show that the set of variable versions newly constrained within the target procedure body are disjoint from the initial variable versions, which we do efficiently using the global variable versions discussed in Subsection 2.5.5. Moreover, we must show that T constrains at most our chosen X w.r.t. the target variable context Λ_t, which follows trivially from our choice of T.

2.5.8. Constant Propagation and Old Expression Desugaring

As discussed, as part of the assignment elimination transformation, the Boogie verifier performs constant propagation and desugars old expressions. We will now discuss how to extend the approach presented so far to incorporate both of these rewrites.

Constant propagation

Boogie's constant propagation removes assignments of a constant literal c to a variable x and then replaces all related occurrences of x by c, before performing the remainder of assignment elimination transformation discussed so far.[28] Related occurrences of x are those that read the value assigned to by the removed assignment; if another assignment happens before the occurrence of x, then the occurrence is not affected. For example, in Figure 2.17 on page 51 the assignment `j := 0` in block B'_0 would be removed, and the occurrences of `j` in **`assert`** A in B'_0 would be replaced by 0. No other occurrences of `j` are affected, since the **`havoc`** in B'_1 updates `j` before any of those occurrences.

28: Boogie does not implement constant propagation as a separate transformation, but performs the constant propagation on-the-fly during assignment elimination.

Extending our approach to generate proofs in the presence of constant propagation is straightforward. We do so by first extending the variable relations $\mathcal{V}_R$ used in our local block lemmas, global block theorems, and in the definitions of expRel and assignElimRel (see Figure 2.18 on page 55 for the original definition) to relations that relate a source variable with a target variable *or* a constant literal (instead of always relating a source variable with a target variable). This extension permits tracking the information needed to justify replacing a variable by a constant literal. The relation between states w.r.t. such an adjusted relation $\mathcal{V}_R$ is lifted in a natural way: if $\mathcal{V}_R$ relates a source variable x with a constant literal c, then the source state must map x to c.

Second, we adjust the signature and definition of assignElimRel to deal with constant propagation. We adjust the parameter for the variable updates to also reflect updates that assign constant literals to variables. Moreover, we add two additional rules for the definition of assignElimRel. One rule captures assignments with constants literals in their right-hand

Figure 2.20: An example showing assignment elimination in the presence of old expressions. g is a global variable. The source CFG is shown on the left and the target CFG is shown on the right.

```
g := g+1
assert old(g+2) > g
```
C_0

```
assume g1 == g0+1
assert g0+2 > g1
```
C_0'

side (instead of arbitrary expressions as already handled by rule ASSIGN-AE in Figure 2.18) and the other rule captures synchronisation commands for variables that are known to map to a constant literal. The generation of proofs for assignElimRel judgements remains simple because even after the addition of the two rules, it is always clear which rule to apply to next.

Desugaring old expressions

Boogie replaces an old expression `old(e)` with `e` where each global variable in `e` is replaced by the corresponding initial variable version in the target, which captures the value that the global variable has at the beginning of a procedure. As an example, consider Figure 2.20, where the procedure consists of a single basic block and `g` is a global variable. The old expression `old(g+2)` is replaced by `g0+2` where `g0` is the initial variable version of `g`.

To handle old expressions in our certification, we track an *initial version mapping* $\mathcal{V}_R^0$ from source global variables to the initial target variable version (in addition to the already-tracked variable relation $\mathcal{V}_R$). We include this mapping in our local block lemmas, global block theorems, and in the definitions of expRel and assignElimRel. In our local block lemmas and global block theorems, we consider states related by both the variable relation $\mathcal{V}_R$ *and* the initial version mapping $\mathcal{V}_R^0$. A source state σ_s and a target state σ_t are related by the initial version mapping $\mathcal{V}_R^0$ if for each global variable g that is mapped to g_0 by $\mathcal{V}_R^0$, the old state component in σ_s maps g to the same value as the value stored for g_0 in σ_t. With this adjusted state relation, it becomes possible to relate old expressions in the source with the desugared expressions in the target.

2.6. VC Generation

In this section, we present our certificate generation approach for the transformation subsequence including the final generation of the VC and the peephole optimisations in the Boogie verifier (this is the certificate connecting procedure P_7 with the VC in Figure 2.1 on page 17). For the sake of presentation, we will focus on the final generation of the VC in this section and at the end briefly discuss the peephole optimisations.

Final VC generation has two main aspects: (1) it encodes and desugars all aspects of the Boogie type system, employing additional uninterpreted functions and axioms to express its properties [67]; program expression elements such as Boogie functions are analogously desugared in terms of these additional uninterpreted functions, creating a nontrivial logical gap between expressions as represented in the VC and those from

[67]: Leino et al. (2010), *A Polymorphic Intermediate Verification Language: Design and Logical Encoding*

the input program, and (2) it performs an efficient (block-by-block) calculation of a weakest precondition for the (acyclic, passified) CFG, resulting in a formula characterising its verification requirements, subject to background axioms and other conditions.

In the following, we will first present the high-level structure of the VC (Subsection 2.6.1) and how Boogie desugars the type system together with the accompanying challenges for our certificate generation (Subsection 2.6.2). Then, we will discuss how to obtain the validity of Boogie's weakest precondition from the validity of the generated VC (Subsection 2.6.3) and how to generate certificates using the validity of Boogie's weakest precondition (Subsection 2.6.4). Finally, we briefly discuss the peephole optimisations (Subsection 2.6.5).

Multiple type encodings

The Boogie verifier implementation contains multiple approaches for the desugaring of Boogie's type system, which can be selected via command-line options. In our work, we support only the main approach that was used by Boogie front-ends that use polymorphic maps and universal type quantification (*e.g.* Dafny and Viper) when we started the work and which was also the default Boogie option for Boogie programs with these features at that time. This option can be forced via the Boogie command-line argument `/typeEncoding:p` and is one of the two approaches introduced by Leino and Rümmer [67]. We discuss one of the unsupported approaches (monomorphisation) in future work in Section 2.12, which has more recently been extended to support polymorphic maps and type quantification, and is now the default option for programs with such features. Viper still uses the type desugaring option that we support in our work. Dafny has initiated experiments with monomorphisation, but still supports the type desugaring option that we support in our work.

2.6.1. VC Structure

The VC generated by Boogie has the following overall structure (represented as a shallow embedding in our certificates):[29]

29: Note that top-level quantification over functions is implicit in the (first-order) SMT problem generated by Boogie; we quantify explicitly in our Isabelle representation.

$$\forall \quad \underbrace{\textit{VC quantifiers}}_{\substack{\text{type encoding parameters,}\\ \text{functions, variable values}}} \quad . \quad (\underbrace{\textit{VC hypothesis}}_{\substack{\text{type encoding,}\\ \text{func./var./prog. axioms}}} \quad \Longrightarrow \quad \textit{CFG WP})$$

The VC quantifies over parameters required for the type encoding, as well as VC counterparts representing the variable values and functions in the Boogie program. The quantifier body is an implication, whose hypothesis (*i.e.* left-hand side) contains: (1) conditions that axiomatise the type encoding parameters, (2) axioms expressing the typing of Boogie variables and functions, and (3) conditions directly relating to axioms explicitly declared in the Boogie program. The conclusion of the implication is an optimised version of the weakest (liberal) precondition (WP) of the CFG.[30]

30: One difference in our version of the Boogie verifier is that we switched off the generation of extra variables introduced to report error executions [78]; these are redundant for programs that do not fail and further complicate the VC structure.

2.6.2. Boogie's Logical Encoding of the Boogie Type System

We first explain Boogie's logical encoding of its own type system. Boogie values and types are represented at the VC level by two uninterpreted carrier sorts V and T, respectively. In particular, this means that types are represented by VC terms of sort T. An uninterpreted function typ^{vc} from V to T maps each value to the representation of its type.

Boogie type constructors are each modelled with an (injective) uninterpreted function C with return sort T and taking arguments (per constructor parameter) of sort T. For example, a type constructor `List` with a single type argument (used, for example, in Figure 2.10 on page 35) is represented by an uninterpreted VC function *List* from T to T. Uninterpreted functions modelling projection functions are also generated for each type constructor (C_i^π for each type argument at position i), *e.g.* mapping the representation of a type `List` t to the representation of type t. Axioms at the VC level are used to reflect the different constraints (*e.g.* injectivity of constructors, projection of types). To reflect that two different constructors generate different types, Boogie uses an uninterpreted VC function *cid* from T to the integers, using axioms to ensure that *cid* maps types obtained via different constructors to different integers. All uninterpreted functions at the VC level (which includes typ^{vc}, constructor functions, projection functions, and *cid*) are universally quantified in the VC structure shown in Subsection 2.6.1.

The following three logical statements show the axioms that Boogie generates at the VC level for the type constructors `ref` and `List _` in Figure 2.10 on page 35:

$$\begin{aligned}&\mathit{cid}(\mathit{ref}) = 4\\ &\forall t : T.\ \mathit{cid}(\mathit{List}(t)) = 3\\ &\forall t : T.\ \mathit{List}_1^\pi(\mathit{List}(t)) = t\end{aligned}$$

The first two axioms ensure that the types obtained via the two type constructors are different. The third axiom ensures that the projection function for the first type argument of `List` is as expected. Moreover, the third axiom also ensures that $\mathit{List}(t_1)$ and $\mathit{List}(t_2)$ are different if $t_1 \neq t_2$, which reflects the fact that the type constructor `List _` is injective.

A consequence of using the carrier sorts V and T is that Boogie must embed the primitive types (*e.g.* integers and Booleans) into these sorts. For example, for the integers, Boogie uses two uninterpreted VC functions: (1) $int2V$, which maps integers to V, and (2) $V2int$, which maps V to integers. Moreover, Boogie uses an uninterpreted VC constant (*i.e.* a nullary uninterpreted function) *IntType* of sort T to represent the integer type in T. Boogie then generates the following VC axioms for these functions for the example in Figure 2.10 on page 35:

$$\begin{aligned}&\forall i : \mathsf{int}.\ V2int(int2V(i)) = i\\ &\forall v : V.\ \mathsf{typ}^{vc}(v) = \mathit{IntType} \Longrightarrow int2V(V2int(v)) = v\\ &\mathit{cid}(\mathit{IntType}) = 0\end{aligned}$$

Here int denotes the built-in integer type supported by SMT solvers. The first two axioms ensure that $int2V$ and $V2int$ are (partial) inverses of each other. The last axiom ensures that the embedded integer type is different from the other types (*e.g.* the references and lists).

Boogie's type system encoding is used in the VC to recover Boogie typing constraints for the untyped VC terms. Recovering the constraints is not always straightforward due to discrepancies between the VC components and the corresponding Boogie components, some of which are due to optimisations performed by Boogie. These discrepancies complicate the validation of VC generation.

For instance, each uninterpreted Boogie function is represented via an uninterpreted VC function, which is untyped in the sense that the function arguments and return value are of sort V.[31] The VC function is then constrained via a VC axiom to have the correct types. For example, the function `cons` in Figure 2.10 on page 35 is represented by a VC-level function $cons^{vc}$ that takes two arguments of sort V and returns a value of sort V. Boogie adds the following axiom constraining the types of $cons^{vc}$:

31: There is an exception: if a formal argument has a primitive type (*e.g.* the integer type), then the VC function's signature also uses the corresponding primitive type at the VC level.

$$\forall x : V, xs : V.\ \mathsf{typ}^{vc}(cons^{vc}(x, xs)) = List(\mathsf{typ}^{vc}(x))$$

There are two discrepancies between the VC-level function $cons^{vc}$ and the semantic interpretation of `cons` (as tracked by the function interpretation in the Boogie semantics). First, $cons^{vc}$ takes two values as input, while the corresponding semantic interpretation of `cons` additionally takes a type as input for the type argument. As a consequence of the first discrepancy, Boogie performs an optimisation: the axiom does not quantify over every possible type. Instead, the axiom represents the type parameter via the type of the first argument (*i.e.* $\mathsf{typ}^{vc}(x)$). Second, $cons^{vc}$ is represented as a *total* function, while the semantic interpretation of the Boogie function `cons` is a *partial* function that is constrained to be defined only in the case when the arguments have the correct types. This discrepancy is also reflected in the above axiom: the axiom requires $\mathsf{typ}^{vc}(cons^{vc}(x, xs))$ to always have the expected type, even if x and xs do not represent reference and list values.[32]

32: After we finished the Boogie certification work, the Boogie developers changed the axiom to require the output to have the correct type only if the arguments have the correct type [79], because not doing so led to issues beyond our supported subset. Nevertheless, for the subset that we support, the original VC axioms are justifiable, as our generated certificates show.

Boogie also optimises away the quantification of types in Boogie expressions in many cases. For example, consider the following axiom from Figure 2.10 on page 35:

```
axiom (forall <T> ::
        (forall x: T, xs: List T :: elem(x, cons(x, xs)))
      );
```

The corresponding VC axiom is given by:

$$\forall x : V, xs : V.\ \mathsf{typ}^{vc}(xs) = List(\mathsf{typ}^{vc}(x)) \Rightarrow elem^{vc}(x, cons^{vc}(x, xs))$$

In this VC axiom, the type quantification has been eliminated. All occurrences of `T` in the Boogie axiom have been translated to $\mathsf{typ}^{vc}(x)$. This optimisation reflects that this particular type quantification is redundant, since `T` can be recovered from the type of x.

2.6.3. Working from VC Validity

Our certificates show that the Boogie program right before the final generation of the VC is correct *if* the generated VC is valid (certifying the validity-checking of the VC by an SMT solver is an orthogonal concern). However, to connect the VC validity back to the program requires first constructing Isabelle-level semantic values to *instantiate* the top-level quantifiers in the VC such that the corresponding VC hypothesis (left-hand side of the VC) can be proved and, thus, the validity of the WP (as represented in the VC) can be deduced (see the structure of the VC in Subsection 2.6.1). Our goal is then to prove that the validity of this WP implies that the corresponding Boogie procedure is correct. To ensure that this proof actually is possible, we must ensure that our instantiation yields a WP whose validity indeed implies the correctness of the Boogie procedure. For example, a top-level VC quantifier modelling a Boogie function f must be instantiated with a mathematical function that behaves in the same way as the semantic interpretation of f for arguments of the correct type.

Our instantiation of the top-level quantifiers depends on the type and function interpretations for the uninterpreted Boogie functions and types (*e.g.* to instantiate a top-level VC quantifier representing a Boogie function), and also depends on an initial Boogie state to instantiate the top-level VC quantifiers representing Boogie variables. Recall that the function interpretation, type interpretation, and initial Boogie state are universally quantified in the procedure correctness (see Subsection 2.3.4 on page 31). When showing that the validity of the VC implies the correctness of the corresponding procedure, we first fix an arbitrary well-formed function interpretation, well-formed type interpretation, and initial well-typed state, such that we can then choose the VC quantifiers using these parameters.

We instantiate the carrier sort V for values in the VC with the corresponding type denoting Boogie values in our formalisation and we instantiate the carrier sort T for types to be all Boogie types that do not contain free variables (*i.e.* closed types). Constructing explicit models for the quantified functions used to model Boogie's type system (satisfying, for example, suitable inverse properties for the projection functions) is straightforward. For example, the VC-level function typ^{vc} that maps each value of V to a type of T is instantiated to be $\lambda v.\ \mathsf{typ}_{\mathcal{T}}(v)$, where $\mathcal{T}$ is the type interpretation and $\mathsf{typ}_{\mathcal{T}}(v)$ returns the type of v w.r.t. $\mathcal{T}$. For the VC-level variable values, we can directly instantiate the corresponding values in the initial Boogie program state.[33]

33: There is only one VC-level variable per Boogie program variable, since the Boogie program considered right before final VC generation has no variable updates and thus every Boogie variable has a single value throughout a single execution.

VC-level functions representing those declared in the Boogie program are instantiated as (total) functions which, *for input values of appropriate type* (the arguments and output are untyped values of sort V), are defined simply to return the same values as the corresponding function in the function interpretation. As discussed in Subsection 2.6.2, perhaps surprisingly, Boogie's VC embedding of functions includes an axiom that requires the VC functions to return values of the specified return type even if the input values do not have the types specified by the function. In such cases, to make sure that this axiom holds, we define the instantiated function to return some value of the specified return type, which is possible since in well-formed type interpretations every closed type has at least one value associated with this type. For example, the instantiation for the function `cons` in Figure 2.10 is given by the following Isabelle function:

$$\begin{aligned} \mathit{cons}^{vc}(x, xs) = \mathsf{case}\ & \mathit{cons}^{interp}([\mathsf{typ}_{\mathcal{T}}(x)], [x, xs])\ \mathsf{of} \\ & \mathsf{Some}(\mathit{res}) \Rightarrow \mathit{res} \\ & \mathsf{None} \Rightarrow \mathsf{SOME}\ v.\ \mathsf{typ}_{\mathcal{T}}(v) = \mathtt{List}(\mathsf{typ}_{\mathcal{T}}(x)) \end{aligned}$$

Here, cons^{interp} is the semantic interpretation of `cons` as tracked in the function interpretation, and $\mathcal{T}$ is the type interpretation. So, our VC instantiation for `cons` returns the same value as the semantic interpretation if the semantic interpretation is defined on the input values and otherwise returns some value whose type is a list. For the former case, we must provide a concrete type to the semantic interpretation cons^{interp} for the type parameter in `cons`, which we do via the type of the first argument. For the latter case, we use Hilbert's epsilon operator (denoted via $\mathsf{SOME}\ v.\ P(v)$ in Isabelle), which picks some value that satisfies the specified predicate (if such a value exists, which it does in this case as discussed above).

After our instantiation, we need to prove the hypothesis of the VC's implication. In particular, we need to prove that all axioms generated by the type system encoding and those coming from the program itself are satisfied. The former are standard and simple to prove (given the work above). The latter (*i.e.* the axioms declared in the Boogie program itself) largely follow from the assumption that each declared axiom must be satisfied in the initial state restricted to the constants. In this latter case, the only remaining challenge is to relate each Boogie axiom (given by a Boogie expression) with the corresponding VC expression in the hypothesis of the VC's implication; a challenge which also arises (and whose solution is explained) below, where we show how to connect the validity of the instantiated WP to the correctness of the corresponding procedure.

Dependent types for the instantiation of the carrier sorts for alternative versions of the Boogie semantics

As an aside, if one were to consider alternative versions of our Boogie semantics formalised in Section 2.3 on page 24, then it turns out that one would need *dependent types* (*i.e.* types that depend on values) in order to accurately instantiate the carrier sorts V and T in the VC, as we will discuss below. Since Isabelle does not support dependent types, this means one would not be able to represent the VC via a shallow Isabelle embedding as we do now (at least not with our current approach of dealing with the VC). A workaround to support such alternative versions of the Boogie semantics would be to represent the VC using a deep embedding in Isabelle instead of a shallow embedding. The satisfiability of a deeply embedded VC would be parameterised by an interpretation of the uninterpreted sorts. This interpretation could then be represented by a mapping from the type used to represents sorts to sets of values of some carrier type (this carrier type would be a type argument of the definition of satisfiability).

One example where one would need dependent types is the following. In Subsection 2.3.3, we discussed an alternative semantics for the quantification of types in Boogie that quantifies over only those Boogie types that are constructible via type constructors declared in the program, instead of considering all possible type constructors (even those not declared in the program). To reflect this alternative semantics directly in the VC, the instantiation of the carrier sort T would have to capture just those Boogie types constructible via the declared type constructors in the program. This *instantiation* of T would thus have be a dependent type at the level of Isabelle, since the instantiation depends on the type constructors declared in the Boogie program.
Another example is if one changes the type interpretation in the Boogie semantics to be a *partial* mapping from abstract values to closed types obtained via type constructors instead of a total mapping. In this case, the instantiation of the carrier type V would have to contain only the primitive values and those abstract values that have a type according to the type interpretation. This instantiation is again a dependent type, since this instantiation depends on the type interpretation.

2.6.4. Certifying the VC Generation

Boogie's weakest precondition calculation is made size-efficient by the usage of explicit named constants for the weakest preconditions $wp(B, \texttt{true})$ for each block B, which is defined in terms of the named constants for its successor blocks. By $wp(B, \texttt{true})$ we mean the weakest precondition of the entire Boogie CFG *starting from* block B w.r.t. postcondition **`true`** (since in this notation here B also represents executions going beyond block B, only postcondition **`true`** is relevant). For example, in Figure 2.17 on page 51, $wp(B_2'', \texttt{true})$ is given by $i_1^{vc} \neq 0 \Longrightarrow wp(B_3'', \texttt{true}) \land wp(B_4'', \texttt{true})$. Here, i_1^{vc} is the value that we instantiated for the quantified VC variable corresponding to the Boogie variable `i1`.

We exploit this modular construction of the generated weakest precondition for the corresponding local and global block theorems. We prove the following local block lemmas for each block:

Theorem 2.6.1 (VC generation local block lemma) *Let B be a block with commands cs in the source CFG G and let Λ be the variable context for the procedure. Let $\mathcal{T}$, $\mathcal{F}$, σ be the type interpretation, function interpretation, and initial Boogie state considered for the proof. Then, for any outcome s', the following must hold: if*

1. $(\mathcal{T}, \Lambda, \mathcal{F}) \vdash \langle cs, \mathsf{N}(\sigma)\rangle \; [\rightarrow] \; s'$
2. $wp(B, \texttt{true})$

then $s' \neq \mathsf{F}$ and if s' is a normal outcome, then

$$\forall B_{suc} \in successors(G, B).\; wp(B_{suc}, \texttt{true})$$

We use the local block lemmas to prove the following global block theorems for each block (in reverse-topological order of the CFG):

Theorem 2.6.2 (VC generation global block theorem) *Let B be a block in the source CFG G and let Λ be the variable context for the procedure. Let $\mathcal{T}$, $\mathcal{F}$, σ be the type interpretation, function interpretation, and initial Boogie state considered for the proof. Then, for any CFG configuration (m', s'), the following must hold: if*

1. $(\mathcal{T}, \Lambda, \mathcal{F}), G \vdash (\mathit{inl}(B), \mathsf{N}(\sigma)) \rightarrow^*_{\mathit{CFG}} (m', s')$
2. $wp(B, \texttt{true})$

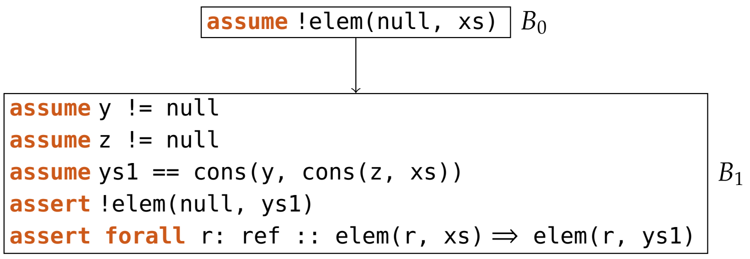
then $s' \neq \mathsf{F}$.

Combining the local block lemmas to obtain the corresponding global block theorems is straightforward. The main challenge is the automatic proof generation for the local block lemma itself for a block B such that Isabelle can automatically check the proof. Our automation approach decomposes the proof by considering each command in B separately, starting from the first command in B and then continuing with the remaining commands iteratively. We outline this decomposition next.

At the point of the final VC generation transformation, each command in a block B must be either an `assume` or an `assert` command. If the first command is `assume` e for some e, we rewrite $wp(B, \texttt{true})$ into the form $e^{vc} \Longrightarrow H$, where e^{vc} is the VC counterpart of e and where H corresponds to the weakest precondition of the remaining commands. This rewriting may involve undoing certain optimisations on the formula structure, which Boogie performs. For instance, Boogie sometimes converts $e^{vc} \Longrightarrow e' \Longrightarrow e'' \Longrightarrow A$ to $(e^{vc} \wedge e') \Longrightarrow (e'' \Longrightarrow A)$; one reason is to decrease the maximum depth of the Boogie implementation's abstract syntax tree representation for formulas. Next, we prove that e evaluates to e^{vc} in the initial Boogie state (we illustrate below how we generate this proof automatically). Hence, if e evaluates to true (*i.e.* the execution does not go to magic), then H must be true because we may assume that $wp(B, \texttt{true})$ holds according to the local block lemma. So, we continue with the same process using the second command in B with H as the corresponding weakest precondition. If the first command is `assert` e, then the approach is similar to the `assume` case: we rewrite $wp(B, \texttt{true})$ to $e^{vc} \wedge H$ (which may again involve undoing optimisations). Thus, both e^{vc} and H must hold. As in the `assume` case, we prove that e evaluates to e^{vc} in the initial Boogie state. Thus, e must evaluate to true (since e^{vc} holds) and hence `assert` e does not fail. We then continue with the same process using the second command in B with H as the corresponding weakest precondition.

In our decomposition, as mentioned, our proof automation rewrites the weakest precondition into a specific form at various points. To simplify this automatic rewriting, we instrument the Boogie verifier such that it produces hints for each command for which the verifier performs optimisations for the corresponding weakest precondition. Our proof automation, expressed via a custom Isabelle tactic, uses these hints to rewrite the weakest precondition into the desired form as specified in the previous paragraph.

To make some of these optimisations and the corresponding hints more concrete, consider Figure 2.21, which illustrates the weakest precondition that Boogie generates for the example in Figure 2.10 on page 35. The



```
assume !elem(null, xs)
```

```
assume y != null
assume z != null
assume ys1 == cons(y, cons(z, xs))
assert !elem(null, ys1)
assert forall r: ref :: elem(r, xs) ⇒ elem(r, ys1)
```

$$wp(B_0, \texttt{true}) \triangleq \neg elem^{vc}(null^{vc}, xs^{vc}) \Rightarrow wp(B_1, \texttt{true})$$

$$wp(B_1, \texttt{true}) \triangleq \begin{array}{l} y^{vc} \neq null^{vc} \Rightarrow \\ z^{vc} \neq null^{vc} \wedge ys_1^{vc} = cons^{vc}(y^{vc}, cons^{vc}(z^{vc}, xs^{vc})) \Rightarrow \\ \neg elem^{vc}(null^{vc}, ys_1^{vc}) \wedge \\ \left[\neg elem^{vc}(null^{vc}, ys_1^{vc}) \Rightarrow \forall r : V.\ (\mathsf{typ}^{vc}(r) = ref \wedge elem^{vc}(r, xs^{vc})) \Rightarrow elem^{vc}(r, ys_1^{vc})\right] \end{array}$$

Figure 2.21: The source CFG (top) on which final VC generation is applied for the example in Figure 2.10 on page 35 and the corresponding weakest preconditions for the blocks (bottom) as computed by the Boogie verifier.

representation of the generated weakest precondition for block B_1 (*i.e.* $wp(B_1, \texttt{true})$) contains two optimisations that our rewritings undo, in order to systematically decompose the weakest precondition.

A first optimisation is that the VC counterparts for the second and third `assume` commands of B_1 are combined into a single conjunction. That is, the weakest precondition from the second command onwards is represented by $(z^{vc} \neq null^{vc} \wedge ys_1^{vc} = \cdots) \Rightarrow H$ instead of the standard representation $z^{vc} \neq null^{vc} \Rightarrow ys_1^{vc} = \cdots \Rightarrow H$. In general, Boogie sometimes combines more than two `assume` commands together like this, and since expressions in `assume` commands themselves may contain conjunctions it is nontrivial to determine how many commands were combined. In such a case, our instrumentation generates a hint for the first `assume` command that is combined indicating how many of the following `assume` commands are combined (*e.g.* in the example, a hint would be generated for `assume z != null`).

A second optimisation is that the weakest precondition from the command `assert !elem(null, ys1)` onwards is represented via a conjunction combined with an implication. That is, the weakest precondition is represented as $\neg elem^{vc}(null^{vc}, ys_1^{vc}) \wedge (\neg elem^{vc}(null^{vc}, ys_1^{vc}) \Rightarrow H)$ instead of the standard representation $\neg elem^{vc}(null^{vc}, ys_1^{vc}) \wedge H$. There are other cases, where this optimisation is not performed.[34] Our instrumentation generates hints for each `assert` command specifying whether this optimisation is performed.

34: The optimisation is called *subsumption* in Boogie and can be disabled via a user-specified attribute on the `assert` command.

Automatically proving that a Boogie expression evaluates to its VC counterpart

For the automatic proof generation of the local block lemma and also for discharging the conditions in the VC hypothesis related to the Boogie program axioms, we need to automatically prove that a Boogie expression

$$\frac{\begin{array}{c}(\mathcal{T},\Lambda,\mathcal{F}),\Omega \vdash \langle e_1,\sigma\rangle \Downarrow \mathsf{BoolVal}({e_1}^{vc}) \\ (\mathcal{T},\Lambda,\mathcal{F}),\Omega \vdash \langle e_2,\sigma\rangle \Downarrow \mathsf{BoolVal}({e_2}^{vc})\end{array}}{(\mathcal{T},\Lambda,\mathcal{F}),\Omega \vdash \langle e_1 \wedge e_2,\sigma\rangle \Downarrow \mathsf{BoolVal}({e_1}^{vc} \wedge {e_2}^{vc})}\ (\textsc{conj-vc})$$

$$\frac{\forall \tau.\ \mathsf{closed}(\tau) \Longrightarrow (\mathcal{T},\Lambda,\mathcal{F}),\Omega(t \mapsto \tau) \vdash \langle e,\sigma\rangle \Downarrow \mathsf{BoolVal}(P(\tau))}{(\mathcal{T},\Lambda,\mathcal{F}),\Omega \vdash \langle \mathbf{forall}\ \langle t\rangle :: e,\sigma\rangle \Downarrow \mathsf{BoolVal}(\forall t : T.P(t))}\ (\textsc{forallt-vc})$$

$$\frac{\forall \tau.\ \mathsf{closed}(\tau) \Longrightarrow (\mathcal{T},\Lambda,\mathcal{F}),\Omega(t \mapsto \tau) \vdash \langle e,\sigma\rangle \Downarrow \mathsf{BoolVal}(e^{vc})}{(\mathcal{T},\Lambda,\mathcal{F}),\Omega \vdash \langle \mathbf{forall}\ \langle t\rangle :: e,\sigma\rangle \Downarrow \mathsf{BoolVal}(e^{vc})}\ (\textsc{forallt-opt-vc})$$

$$\frac{\begin{array}{c}\mathsf{closed}(\mathsf{substT}(\Omega,\tau)) \\ \forall w.\ \mathsf{typ}_{\mathcal{T}}(w) = \mathsf{substT}(\Omega,\tau) \Rightarrow (\mathcal{T},\Lambda,\mathcal{F}),\Omega \vdash \langle e,(os,gs,ls(x \mapsto w))\rangle \Downarrow \mathsf{BoolVal}(P(w)) \\ \forall w. \neg P(w) \Rightarrow \mathsf{typ}_{\mathcal{T}}(w) = \mathsf{substT}(\Omega,\tau)\end{array}}{(\mathcal{T},\Lambda,\mathcal{F}),\Omega \vdash \langle \mathbf{forall}\ x : \tau :: e,(os,gs,ls)\rangle \Downarrow \mathsf{BoolVal}(\forall v : V.\ P(v))}\ (\textsc{forall-vc})$$

Figure 2.22: Selected syntax-directed rules to relate a Boogie expressions with a VC-level expression.

e evaluates in a state σ to its VC counterpart e^{vc}, which is formally expressed by the following statement:

$$(\mathcal{T},\Lambda,\mathcal{F}),\Omega \vdash \langle e,\sigma\rangle \Downarrow \mathsf{BoolVal}(e^{vc}) \qquad (2.3)$$

Note that e^{vc} is a shallowly embedded Isabelle formula that includes the instantiations of quantified variables, which we constructed above. Hence, e^{vc} is a Boolean value in Isabelle, and thus must be wrapped by the Boogie value constructor $\mathsf{BoolVal}(\cdot)$ in order to be lifted to a Boogie value.

Our approach for automatically proving properties such as statement 2.3 works largely on syntax-driven rules. Our approach must account for the discrepancies between the Boogie program and the VC representation such as the mismatching function signatures, and the adjustments and optimisations that Boogie made either to the formula structure or via the type system encoding (see Subsection 2.6.2). We handle some of these discrepancies by rewriting the formula into a standard form that we require for our syntax-driven rules or by capturing such a discrepancy directly in a syntax-driven rule, and in other cases we directly work with the provided formula. All cases are automated using Isabelle tactics.

Figure 2.22 shows some of the syntax-directed rules that our proof automation applies to prove that a Boogie expression evaluates to its VC counterpart. Rule conj-vc relates a Boogie conjunction with a VC conjunction by splitting the evaluation into the two conjuncts. There are two rules for relating universal type quantifications: rule forallt-vc handles the case when the type quantification is not optimised away in the VC, while rule forallt-opt-vc handles the case when it is optimised away (we showed an example where such an optimisation is applied in Subsection 2.6.2 on page 64).

Rule forall-vc relates universal value quantifications for non-primitive types (the corresponding rule for primitive types is simpler). The first two premises require that (1) the quantified type has no free variables after substituting the type variables tracked by Ω and (2) the quantifier bodies are related for any value of the specified type. The third premise requires that if the VC quantifier body does not hold for some quantified value, then it must be the case that the quantified value has the specified type. This third premise is necessary to ensure that if the VC quantification is false, then the witness showing this negative result can be mapped back to a witness showing that the Boogie quantification evaluates to false. In practice, $P(w)$ is of the form $\mathsf{typ}^{vc}(w) = \cdots \Rightarrow e^{vc}$, so its negation satisfies $\mathsf{typ}^{vc}(w) = \cdots$, which is sufficient to prove the third premise. Moreover, note that $\mathsf{typ}^{vc}(w) = \cdots \Rightarrow e^{vc}$ is syntactically not in sync with the Boogie quantifier body e due to the left-hand side $\mathsf{typ}^{vc}(w) = \cdots$. As a result, our Isabelle tactic, when proving the second premise of FORALL-VC, rewrites $\mathsf{typ}^{vc}(w) = \cdots \Rightarrow e^{vc}$ to e^{vc} (by proving $\mathsf{typ}^{vc}(w) = \cdots$) before proceeding, which is possible due to the hypothesis in the second premise expressing that the quantified value w has the specified type.

Our Isabelle tactic applies our syntax-directed rules until a Boogie expression and corresponding VC expression are reached for which none of the rules apply. At this point, our tactic instead uses the expression evaluation rules shown in Figure 2.4 on page 29 and Figure 2.5 on page 30. Working with the standard expression evaluation rules leads to more complex proof obligations, because these rules do not constrain the shape of the corresponding VC expression. While our tactic is still able to prove many examples where such more complex proof obligations arise, there are examples where our tactic cannot complete the proof because the Isabelle tactic that we apply at particular points is not able to solve the resulting proof goals. In particular, we have experienced incompletenesses for certain function call cases, because we currently do not have a syntax-directed rule for function calls, and thus our tactic applies the standard expression evaluation rule for function calls. As our evaluation (Section 2.10) shows, examples taken from the Boogie test suite where our automation fails are rare, but such examples could appear more frequently when considering Boogie programs from other sources. Fixing our automation to deal with these cases via more syntax-directed rules should be conceptually straightforward, but will require some nontrivial engineering effort since one must distinguish between different cases for function calls. We discuss one particular function call case where our tactic does not work well, and briefly discuss how one could fix our tactic to deal with such cases.

For instance, our tactic does not work well in cases where the VC uses conversion functions to transform function call values of the carrier type V to a primitive type. Such conversions occur, for instance, in calls to a polymorphic function f that takes a type argument T and that returns a value of type T. For instance, consider the Boogie function

```
function hd<T>(xs: List T) : T
```

A function call `hd(xs)`, where `xs` is of type `List int`, is in some contexts translated to the VC function call $V2int(hd^{vc}(xs))$, where $V2int(\cdot)$ converts a value of the carrier type V to an integer at the VC level. To handle such an occurrence, we would have to prove that $hd^{vc}(xs)$ returns a value in V that corresponds to an integer, which requires proving that xs indeed has type `List int`. Our automation currently does not do this. A solution would be to have a syntax-directed rule for function calls where the automation uses different approaches depending on whether a conversion occurs; in the case of a conversion, the automation would prove the necessary typing constraints. Such an extension to our automation must distinguish different cases (*e.g.* whether a conversion occurs or not) and must incorporate proofs showing that Boogie expressions are well-typed (and thus the corresponding VC values also have the expected type) to prove the necessary typing constraints. For the latter, we already have automation (proving that a Boogie expression is well-typed), but one still needs to incorporate this type constraint reasoning into the overall tactic.

2.6.5. Peephole Optimisations

Peephole optimisations, which are also included as part of the final VC generation certificate, prune unreachable blocks and remove empty blocks. Dealing with these transformations is straightforward. We will discuss what the pruning of unreachable blocks does and how to validate it when discussing the CFG optimisations (see Subsection 2.7.4 on page 79), which contain the same pruning transformation. We discuss our validation of the removal of empty blocks next.

When an empty block B is removed, there is no corresponding weakest precondition for B, and so we cannot directly use the approach discussed above. Instead, we identify the set of all non-empty blocks E that are reachable from B by visiting only empty blocks first. The blocks in E are the first blocks reached from B that are not removed. As a result, the blocks in E are captured in the generated weakest precondition and the weakest precondition of B is captured in the generated weakest precondition by the conjunction of the weakest preconditions of the blocks in E. Therefore, we show a global block theorem for B where the weakest precondition in the premise is the conjunction of the weakest preconditions of E. Such global block theorems are straightforward to prove and allow the rest of the approach to work as presented.

This concludes our discussion of the certification of Boogie's three key transformations: cycle elimination, assignment elimination, and final VC generation. In the following sections, we will discuss the remaining two transformations in the pipeline: the CFG optimisations, and the AST-to-CFG transformation.

2.7. CFG Optimisations

In this section, we present our certificate generation approach for the CFG optimisations that the Boogie verifier performs right after the AST-to-CFG transformation (in Figure 2.1 on page 17, the CFG optimisations are shown between P_2 and P_3). The CFG optimisations, for our supported Boogie subset, consist of three separate transformations: (1) a subset of unreachable blocks are pruned, (2) blocks are coalesced to reduce the number of blocks, (3) dead variables are eliminated.

As discussed in Section 2.2, our generated certificates support only (1) and (2). Figure 2.23 illustrates this. We generate a single certificate that

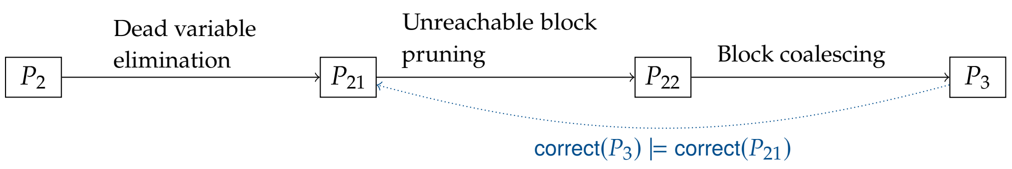

Dead variable elimination
Unreachable block pruning
Block coalescing
P_2
P_{21}
P_{22}
P_3
correct(P_3) $\models$ correct(P_{21})

Figure 2.23: The transformations applied by Boogie as part of the CFG optimisations on source procedure P_2 resulting in a target procedure P_3. Our generated certificates currently capture unreachable block pruning and block coalescing (dotted edge in blue), but not dead variable elimination. If there are no dead variables, then $P_2 = P_{21}$ and as a result, we are able to validate the entire Boogie pipeline when combining the certificate shown by the dotted edge here with the certificates for the other transformations.

covers both (1) and (2) (analogously to how our generated certificate for cycle elimination covers multiple transformations). If there are no dead variables (and thus (3) has no effect), then we can combine the certificate for (1) and (2) with the generated certificates for the other transformations to validate the entire Boogie pipeline. Otherwise, we combine the generated certificates to cover the entire Boogie pipeline excluding the elimination of dead variables. Concretely, in this latter case, our combined certificate shows that the validity of the VC implies the correctness of the procedure after dead variable elimination (but before the other two optimisations; the corresponding procedure is P_{21} in Figure 2.23) and a certificate showing the soundness of the AST-to-CFG transformation. That is, one must currently trust that dead variable elimination is correct if dead variables exist (*i.e.* correctness of P_{21} implies correctness of P_2 in Figure 2.23).

There are two main differences when formally validating the CFG optimisations compared to the previously discussed transformations: (1) in the CFG optimisations, *both* the source and the target CFG can be cyclic, and (2) due to the coalescing of blocks, a single block in the target CFG must be related to *multiple* blocks in the source CFG. We will first discuss how we validate the coalescing of blocks, which shows how to deal with both of these differences. At the end, we discuss how to deal with the pruning of unreachable blocks. More details can be found in Lukas Himmelreich's BSc thesis [62], which was supervised by the author of this dissertation.

[62]: Himmelreich (2023), *Formally Validating the CFG Optimization Phase of the Boogie Program Verifier*

2.7.1. Validation Approach for Block Coalescing

Boogie coalesces two blocks B_0 and B_1 into a single block if B_0 has B_1 as its only successor and B_1 has B_0 as its only predecessor. In general, Boogie applies coalescing until it is no longer possible, which means that an arbitrary large sequence of blocks may be coalesced into a single block. The commands themselves remain the same. Figure 2.24 shows a concrete example where block coalescing has an effect. Here, the source blocks B_0, B_1, and B_2 are coalesced into the target block B'_0, and the source blocks B_5 and B_6 are coalesced into the target block B'_5.

For the validation of block coalescing, we do not define separate local block lemmas relating source and target commands of single blocks, since the commands remain unchanged; it is sufficient to just check whether the commands in the source and target match. However, we still define global block theorems that relate source and target executions

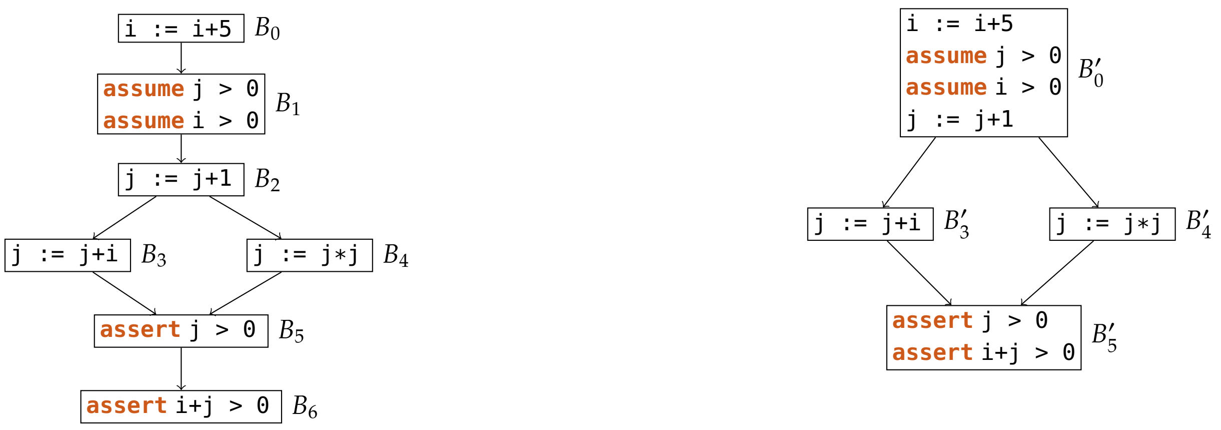


Figure 2.24: Example showing the application of block coalescing to the source CFG on the left, which results in the target CFG on the right.

throughout the CFG. A difference to the previous transformations is that in the case when a source block is coalesced, we cannot phrase the global block theorem by relating executions starting from the source block and executions starting from the corresponding target block B_T. Instead, we need to relate executions starting from the source block with executions starting from the *corresponding intermediate command* in B_T. For instance, executions that begin in the coalesced block B_1 in Figure 2.24 should be related with executions that begin right before command `assume j > 0` in block B'_0.

The second difference compared to the previously discussed transformations is that both the source and target CFGs may be cyclic. For cycle elimination (Section 2.4), we presented a validation approach to deal with cyclic source CFGs and acyclic target CFGs. For the CFG optimisations, we use a similar approach as for cycle elimination to deal with cycles. To prove a global block theorem for a loop head, we perform an induction proof. Global block theorems for blocks within loops have as premises the corresponding induction hypotheses of each loop that the block is contained in. When proving a global block theorem for a block that is an origin of a back edge that goes back to a loop head B_H, we can use the corresponding induction hypothesis to justify executions that go back to B_H.

A difference to the validation of cycle elimination is how we obtain the order in which the global block theorems are proved such that a global block theorem for a block can be proved using the theorems of the corresponding successors (*i.e.* proving the global block theorems without inducing any circular dependencies between the global block theorems). For cycle elimination, we were able to use the reverse-topological order of the target CFG since the target CFG was acyclic. For the CFG optimisations, we do something similar conceptually: we use the reverse-topological order of the target CFG *without* its back edges (*i.e.* edges from within a loop back to the loop head). Since Boogie already computes the back edges of the target CFG to perform cycle elimination, we need not recompute the back edges.

2.7.2. Global Block Theorems

To make our validation approach more concrete, we will now show the formal definitions of the global block theorems that we automatically generate and prove (one per source block). We distinguish between two kinds of global block theorems: (1) a *global standard block theorem* that we prove if the source block is not coalesced and (2) a *global coalesced block theorem* that we prove if the source block is coalesced.

Global standard block theorem

The global standard block theorem is given by:

Theorem 2.7.1 (CFG optimisations global standard block theorem) *Let Λ be the variable context of the source and target procedure. Let Q_{post} be the postcondition of the procedure. Let G_S be the source CFG and let G_T be the target CFG. Let B_S be a source block and let B_T be the corresponding target block. For any source block B, let $\mathsf{LH}(B)$ be the set of loop heads of the loops that B is contained in (excluding B if B is a loop head). For any source loop head B, let $\mathsf{Trg}(B)$ be the corresponding block in the target CFG.*

Then, for any type interpretation $\mathcal{T}$, any function interpretation $\mathcal{F}$, any state σ, any CFG configuration (m_1', s_1'), and any number of execution steps i the following must hold: if

1. $(\mathcal{T}, \Lambda, \mathcal{F}), G_S \vdash (\mathsf{inl}(B_S), \mathsf{N}(\sigma)) \rightarrow^i_{\mathsf{CFG}} (m_1', s_1')$
2. $\forall m_2', s_2'.\ (\mathcal{T}, \Lambda, \mathcal{F}), G_T \vdash (\mathsf{inl}(B_T), \mathsf{N}(\sigma)) \rightarrow^*_{\mathsf{CFG}} (m_2', s_2') \implies \mathsf{validConfig}(\mathcal{T}, \Lambda, \mathcal{F}, Q_{post}, (m_2', s_2'))$
3. $\forall B_H \in \mathsf{LH}(B_S).$ $\mathsf{loopIHOpt}(\mathcal{T}, \Lambda, \mathcal{F}, \Omega, (G_S, G_T), (B_H, \mathsf{Trg}(B_H)), i, Q_{post}, (m_1', s_1'))$

then $\mathsf{validConfig}(\mathcal{T}, \Lambda, \mathcal{F}, Q_{post}, (m_1', s_1'))$ holds

This global standard block theorem is essentially a simpler version of the global block theorem used for cycle elimination (Theorem 2.4.2 on page 45). One difference is that for the CFG optimisations, the absence of failing target executions (second premise) must take the postcondition into account, since the postcondition is inserted into the CFG only later.

Another difference is that the induction hypothesis $\mathsf{loopIHOpt}$ tracked for each loop in the premises is defined differently. The definition of $\mathsf{loopIHOpt}$ is given by:

Definition 2.7.1 ($\mathsf{loopIHOpt}$) *Let B_H be a loop head in the source CFG G_S and let B_H' be the corresponding loop head in the target CFG G_T. Then, $\mathsf{loopIHOpt}(\mathcal{T}, \Lambda, \mathcal{F}, \Omega, (G_S, G_T), (B_H, B_H'), i, (m_1', s_1'), Q_{post})$ holds iff for all j where $j \leq i$ and all states σ the following holds: if*

1. $(\mathcal{T}, \Lambda, \mathcal{F}), G_S \vdash (\mathsf{inl}(B_H), \mathsf{N}(\sigma)) \rightarrow^j_{\mathsf{CFG}} (m_1', s_1')$
2. $\forall m_2', s_2'.\ (\mathcal{T}, \Lambda, \mathcal{F}), G_T \vdash (\mathsf{inl}(B_H'), \mathsf{N}(\sigma)) \rightarrow^*_{\mathsf{CFG}} (m_2', s_2') \implies \mathsf{validConfig}(\mathcal{T}, \Lambda, \mathcal{F}, Q_{post}, (m_2', s_2'))$

then $\mathsf{validConfig}(\mathcal{T}, \Lambda, \mathcal{F}, Q_{post}, (m_1', s_1'))$ holds.

This definition directly reflects the global standard block theorem, except that there is no premise containing the induction hypothesis for each of the loops in which the loop head B_H is contained. The reason for omitting this premise is the same as discussed for cycle elimination's induction hypothesis (Definition 2.4.2 on page 46): the corresponding premise in the global standard block theorem for B_H is sufficient to ensure the hypotheses hold.

One difference to the cycle elimination transformations's induction hypothesis is that the premise requiring the absence of failing executions in the target CFG must be included here in Definition 2.7.1, while it could be omitted for the cycle elimination case. The reason it must be included here is the following. For the CFG optimisations, loop iterations in the source CFG are captured by corresponding loop iterations in the target CFG. That is, given a source loop execution from a fixed state σ, there is a target loop execution that captures the source execution. This target execution reaches potentially *different* states after each loop iteration (namely those states reached by the source execution). The premises in the loop head's global block theorem do not directly imply that there are no failing loop executions in the target beginning from these different states. As a result, we must add the absence of failing target executions as a premise in the induction hypothesis here. On the other hand, in the case of cycle elimination, all target executions justifying the corresponding loop iterations start from the *same* state related to σ, which is why the absence of failing target executions can be justified via the premises in the loop head's global block theorem.

Global coalesced block theorem

To introduce the global coalesced block theorem, we first introduce some necessary terminology. For a source block B_S that is coalesced into a target block B_T, we call the *suffix commands of B_T w.r.t. B_S* the suffix of commands of B_T that captures the intermediate point in B_T reflecting the beginning of B_S (in particular, the commands of B_S must be a prefix of this suffix). For instance, in Figure 2.24 the suffix commands of B'_0 w.r.t. B_2 are given by the singleton list [`j := j+1`] and the suffix commands of B'_0 w.r.t. B_1 are given by the list [`assume j > 0`, `assume i > 0`, `j := j+1`].

The global coalesced block theorem is given by:

Theorem 2.7.2 (CFG optimisations global coalesced block theorem) *Let Q_{post}, Λ, G_S, G_T, B_S and $\mathit{LH}(\cdot)$ and $\mathit{Trg}(\cdot)$ be as in the global standard block theorem (Theorem 2.7.1). Let the source block B_S be coalesced into a target block B_T, and let cs_T be the suffix commands in B_T w.r.t. B_S.*

Then, for any type interpretation $\mathcal{T}$, any function interpretation $\mathcal{F}$, any state σ, any CFG configuration (m'_1, s'_1), and any number of execution steps i the following must hold: if

1. $(\mathcal{T}, \Lambda, \mathcal{F}), G_S \vdash (\mathit{inl}(B_S), N(\sigma)) \rightarrow^i_{CFG} (m'_1, s'_1)$
2. $\mathit{succsCorrectAfterCmds}(\mathcal{T}, \Lambda, \mathcal{F}, G_T, B_T, cs_T, \sigma, Q_{post})$
3. $\forall B_H \in \mathit{LH}(B_S)$.
   $\mathit{loopIHOpt}(\mathcal{T}, \Lambda, \mathcal{F}, \Omega, (G_S, G_T), (B_H, \mathit{Trg}(B_H)), i, Q_{post}, (m'_1, s'_1))$

then $\mathit{validConfig}(\mathcal{T}, \Lambda, \mathcal{F}, Q_{post}, (m'_1, s'_1))$ holds.

$$
\begin{aligned}
&\mathsf{succsCorrectAfterCmds}(\mathcal{T}, \Lambda, \mathcal{F}, G_T, B_T, cs_T, \sigma, Q_{post}) \triangleq \\
&\quad \forall s'.\ (\mathcal{T}, \Lambda, \mathcal{F}) \vdash \langle cs_T, \mathsf{N}(\sigma)\rangle\ [\rightarrow]\ s' \Rightarrow \\
&\qquad s' \neq \mathsf{F} \wedge \big(\mathsf{successors}(G_T, B_T) = \emptyset \Rightarrow \mathsf{validConfig}(\mathcal{T}, \Lambda, \mathcal{F}, Q_{post}, (\mathsf{inr}(), s'))\big) \wedge \\
&\qquad \forall \sigma_1.\ s' = \mathsf{N}(\sigma_1) \Rightarrow \forall B_{succ} \in \mathsf{successors}(G_T, B_T) \Rightarrow \\
&\qquad\quad \forall m_2', s_2'.\ (\mathcal{T}, \Lambda, \mathcal{F}), G_T \vdash (\mathsf{inl}(B_{succ}), \mathsf{N}(\sigma_1)) \rightarrow^*_{\mathsf{CFG}} (m_2', s_2') \Longrightarrow \mathsf{validConfig}(\mathcal{T}, \Lambda, \mathcal{F}, Q_{post}, (m_2', s_2'))
\end{aligned}
$$

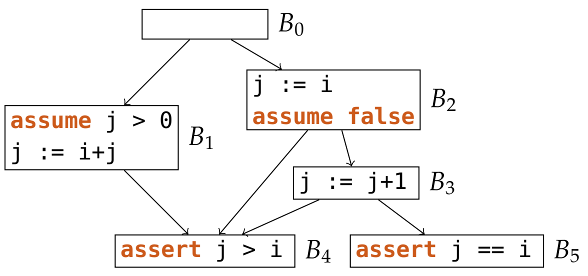
Figure 2.25: Definition of succsCorrectAfterCmds. Note that if block B_T has no successors, then the postcondition must hold after the execution of commands cs_T (except if a magic state is reached); this explains the extra conjunct with the hypothesis $\mathsf{successors}(G_T, B_T) = \emptyset$.

Here, the second premise expresses via succsCorrectAfterCmds that there are no failing executions that start by executing the suffix commands cs_T followed by any execution in the target CFG starting from one of the successors of the target block B_T. (The formal definition for succsCorrectAfterCmds is shown in Figure 2.25.) This captures for a coalesced source block precisely the matching executions that execute the same commands in the target CFG. This second premise is also the only difference to the global standard block theorem (Theorem 2.7.1).

2.7.3. Generating Proofs for Global Block Theorems

As previously mentioned, we automatically generate proofs for the global block theorems introduced in Subsection 2.7.2 in reverse-topological order of the target CFG where the back edges have been eliminated. Since Boogie already computes the target CFG's back edges for cycle elimination, we reuse these results to compute the reverse-topological order. Moreover, we also reuse the back edge results to compute the set of loops in which a source block is contained, which we require to express the global block theorems. This is possible since block coalescing does not fundamentally change the loops. There are some corner cases one must take into account. For example, a loop head in the source CFG may be coalesced together with the entire loop body into a single block. In this case, in the target CFG, the loop is a single block with an edge to itself, while in the source CFG the loop head and the origin of the back edge are different blocks.

To validate the coalescing of blocks, we instrument the Boogie verifier to produce information indicating which source blocks are coalesced into which target block. This avoids having to redo the coalescing computation. Using this information, we generate the global block theorems for a sequence of blocks $(B_0, B_1, ..., B_n)$ coalesced into a single target block as follows. For the final block B_n in the sequence, we prove the global coalesced block theorem using the global standard block theorems for B_n's successors. Then, we iteratively prove the global coalesced block theorems for B_i using the global coalesced block theorem for its unique successor B_{i+1}. Finally, we convert the global coalesced block theorem for B_0 to the corresponding global standard block theorem that relates B_0 with the target block B_0' into which the block sequence was coalesced. To complete the proof in this last step, we must check that the commands of the target block B_0' are given by the commands in the sequence of source blocks $(B_0, B_1, ..., B_n)$.



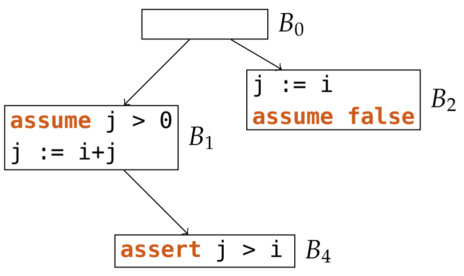


Figure 2.26: Example showing the pruning of unreachable blocks applied to the source CFG on the left, which results in the target CFG on the right.

2.7.4. Unreachable Block Pruning

To prune a subset of blocks unreachable from the entry block in a CFG, Boogie identifies every block that syntactically contains at least one **assume false** or **assert false** command. Let us call such blocks *abnormal blocks*, since every execution of such a block either ends in a magic state (if **assume false** is executed) or in failure (if **assert false** is executed). In the unreachable block pruning transformation, Boogie removes all outgoing edges from abnormal blocks and prunes every block that is reachable from the entry block only via paths that contain at least one abnormal block. This transformation is sound because no execution ever continues beyond an abnormal block. One advantage of block pruning is that the considered CFG becomes smaller. Another advantage is that it enables more blocks being coalesced in the subsequent block coalescing transformation.

Figure 2.26 shows a concrete example of the transformation. Here, blocks B_3 and B_5 are pruned, since every path reaching them must go through the abnormal block B_2. The edge from B_2 to B_4 is pruned, too. In this example, the pruning of blocks leads to B_1 and B_4 being coalesced later.

It is straightforward to extend the validation discussed for block coalescing to include the pruning of unreachable blocks. For abnormal blocks that are not pruned, we directly prove the global block theorem (the standard or coalesced version depending on whether the block is coalesced) without needing to take successors into account; we just need to show that no execution will continue beyond the abnormal block. We do not need to prove global block theorems for pruned blocks for the following reason: If the transformation was performed correctly, then we are able to prove the global block theorem of the entry block without considering any of the pruned blocks. This implicitly shows that no execution from the entry block ever reaches pruned blocks.

2.8. A Formal Semantics For Boogie Abstract Syntax Trees

So far, we have discussed the formal validation of Boogie transformations that operate solely on the CFG representation of Boogie programs. However, the original representation into which a Boogie program is parsed is an AST, which is then converted into a CFG in the first transformation

$$
\begin{aligned}
\mathit{BGuard} \ni g &::= e \mid * \\
\mathit{BControlFlow} \ni \mathit{ctrl} &::= \texttt{if}\ (g)\ \{\ s\ \}\ \texttt{else}\ \{\ s\ \} \mid \texttt{while}\ (g)\ \texttt{invariant}\ e\ \{\ s\ \} \mid \texttt{return} \mid \epsilon \\
\mathit{BStmtBlock} \ni b &::= \overrightarrow{c}; \mathit{ctrl} \\
\mathit{BStmt} \ni s &::= \overrightarrow{b}
\end{aligned}
$$

Figure 2.27: The Boogie AST statement syntax of our formalised Boogie subset. The symbols e and c denote Boogie expressions and basic commands, respectively, as defined in Figure 2.3 on page 25. The symbol ϵ denotes an empty control-flow element.

of Boogie's pipeline (as discussed in Section 2.2). Moreover, Boogie front-end verifiers that translate a source language into Boogie usually directly interact with an AST representation of Boogie. As a result, the formal validation of both of these translations (Boogie's AST-to-CFG transformation and translations from source languages into Boogie) requires a formal semantics for Boogie's AST representation. In this section, we present such a formal semantics.

2.8.1. The Boogie AST

The bodies of Boogie procedures are the only construct in a Boogie program where there is a difference between an AST and a CFG representation. The top-level commands, as defined in Figure 2.3 on page 25, remain the same. An AST procedure body is given by a statement whose syntax is shown in Figure 2.27. A statement is given by a list of *statement blocks*. Each statement block $\overrightarrow{c}; \mathit{ctrl}$ consists of a list of basic commands $\overrightarrow{c}$ followed by a control-flow element *ctrl* that is an if-construct, a while-construct, a return-construct (to exit a procedure prematurely), or is empty (ϵ). The bodies of if- and while-constructs are again statements, and while-constructs are given an invariant as a Boogie expression.[35] We make sure that in our Isabelle embedding of a Boogie program, these bodies are always non-empty, which avoids the need for special cases when defining its semantics. An empty body in the source code is represented via the singleton list with an empty statement block (*i.e.* $[[]; \epsilon]$).

35: Boogie supports the declaration of multiple invariants for a single loop, whose conjunction represents the actual invariant. We also support multiple invariant declarations in our Isabelle formalisation.

The conditions for if- and while-constructs are given by guards that are either Boolean expressions or wildcard symbols (*). Wildcard symbols express demonic nondeterminism. For instance, an if-construct with the wildcard symbol expresses that an execution nondeterministically chooses which branch to check. Correctness of a Boogie program holds only if neither branch leads to failure.

The AST syntax in Figure 2.27 reflects the representation used by the Boogie verifier in its implementation. This representation is somewhat non-standard, since it distinguishes between two different sequential compositions: sequential composition of statement blocks and sequential composition of a list of basic commands followed by a control-flow element. As we will see in Chapter 3, this leads to a mismatch between the structure of the Boogie AST and the more standard Viper AST, which uses only one kind of sequential composition. In Chapter 3, we bridge this mismatch.

We do not formalise all AST constructs that the Boogie verifier supports. For instance, we currently do not formalise breaks out of loops, gotos, and else-if-branches in if-constructs. Else-if branches can be encoded into our formalised subset via nested if-constructs. We also do not formalise Boogie's *free loop invariants* as part of while loops (loop invariants that are assumed to hold without being checked). When unformalised AST constructs are used, then the AST-to-CFG transformation is not validated. However, as we will make clear in Section 2.9, in such cases, our formal validation for the CFG transformations still works if the resulting CFG falls into our formalised CFG subset. For the mentioned unformalised AST constructs, this is indeed the case, since we support CFGs generated from source programs with breaks, gotos, else-if-branches, and free loop invariants.

2.8.2. Operational Semantics

We define a small-step operational semantics for the Boogie AST. The judgement $\Gamma \vdash \langle \gamma, s\rangle \rightarrow_{\mathsf{AST}} \langle \gamma', s'\rangle$ expresses a single execution step from *program point* γ and outcome s to program point γ' and outcome s' in the Boogie context Γ. A program point is given by a pair of the currently active statement block b and the continuation representing the statement blocks to be executed after b. A continuation is either the empty continuation **KStop** (*i.e.* nothing to execute) or a sequential continuation **KSeq**$(b, \mathcal{K})$ (*i.e.* a statement block b followed by a continuation $\mathcal{K}$). A continuation-based small-step semantics avoids the need for local search rules commonly required in a small-step semantics [80].[36]

[80]: Appel et al. (2007), *Separation Logic for Small-Step cminor*

36: Such a continuation-based small-step semantics has also been used with breaks and gotos, which Boogie also has, but which we do not yet support. An example is a semantics used in the CompCert formalisation [81].

The small-step judgement is defined inductively via the rules shown in Figure 2.28. The rules are mostly standard. The semantics for the loop executes each loop iteration separately, where the corresponding invariant is checked before every loop iteration. If an invariant check fails, failure is reached.

As can be seen in the premise of the first rule shown in Figure 2.28, the list of basic commands at the beginning of a statement block execute fully in a single step (this is because of the judgement $\Gamma \vdash \langle cs, \mathsf{N}(\sigma)\rangle \; [\rightarrow] \; s'$ defined in Figure 2.7 on page 31). This is consistent with how the CFG semantics is defined in Subsection 2.3.3, where the list of basic commands for a basic block executes in a single step. However, in Chapter 3, when reasoning about the Viper-to-Boogie translation, it is more useful to use an AST semantics where each basic command in the list of a statement block executes in a separate step. Such an alternative semantics allows expressing more fine-grained reductions, which is useful, for instance, when a Viper statement corresponds only to a prefix of the list of basic commands. As a result, we define an auxiliary semantics that does precisely this via the judgement $\Gamma \vdash \langle \gamma, s\rangle \rightarrow_{\mathsf{AST2}} \langle \gamma', s'\rangle$, which is the same as $\Gamma \vdash \langle \gamma, s\rangle \rightarrow_{\mathsf{AST}} \langle \gamma', s'\rangle$ except that each basic command in a list executes in a separate step. We define this auxiliary semantics in terms of the original one via the two separate rules shown in Figure 2.29. This auxiliary semantics does not show up in our final certificates (and thus need not be trusted), since in the end everything is connected via the original AST semantics.

$$\frac{\begin{array}{c}\Gamma \vdash \langle cs, \mathsf{N}(\sigma)\rangle \; [\rightarrow] \; s' \\ cs \neq []\end{array}}{\Gamma \vdash \langle (cs; ctrl, \mathcal{K}), \mathsf{N}(\sigma)\rangle \rightarrow_{\mathsf{AST}} \langle ([]; ctrl, \mathcal{K}), s'\rangle}$$

$$\frac{}{\Gamma \vdash \langle ([]; \epsilon, \textbf{KSeq}(b, \mathcal{K})), \mathsf{N}(\sigma)\rangle \rightarrow_{\mathsf{AST}} \langle (b, \mathcal{K}), \mathsf{N}(\sigma)\rangle}$$

$$\frac{\begin{array}{c}s = \mathsf{M} \lor s = \mathsf{F} \\ \neg(cs = [] \land ctrl = \epsilon \land \mathcal{K} = \textbf{KStop})\end{array}}{\Gamma \vdash \langle (cs; ctrl, \mathcal{K}), s\rangle \rightarrow_{\mathsf{AST}} \langle ([]; \epsilon, \textbf{KStop}), s\rangle}$$

$$\frac{}{\Gamma \vdash \langle ([]; \texttt{return}, \mathcal{K}), \mathsf{N}(\sigma)\rangle \rightarrow_{\mathsf{AST}} \langle ([]; \epsilon, \textbf{KStop}), \mathsf{N}(\sigma)\rangle}$$

$$\frac{g \neq * \Longrightarrow \Gamma, \emptyset \vdash \langle g, \mathsf{N}(\sigma)\rangle \Downarrow \mathsf{BoolVal(true)}}{\Gamma \vdash \langle ([]; \texttt{if}\ (g)\ \{\ b :: bs\ \}\ \texttt{else}\ \{\ bs'\ \}, \mathcal{K}), \mathsf{N}(\sigma)\rangle \rightarrow_{\mathsf{AST}} \langle (b, \mathsf{blocksToCont}(bs, \mathcal{K})), \mathsf{N}(\sigma)\rangle}$$

$$\frac{g \neq * \Longrightarrow \Gamma, \emptyset \vdash \langle g, \mathsf{N}(\sigma)\rangle \Downarrow \mathsf{BoolVal(false)}}{\Gamma \vdash \langle ([]; \texttt{if}\ (g)\ \{\ bs'\ \}\ \texttt{else}\ \{\ b :: bs\ \}, \mathcal{K}), \mathsf{N}(\sigma)\rangle \rightarrow_{\mathsf{AST}} \langle (b, \mathsf{blocksToCont}(bs, \mathcal{K})), \mathsf{N}(\sigma)\rangle}$$

$$\frac{\begin{array}{c}b_{while} = []; \texttt{while}\ (g)\ \texttt{invariant}\ e\ \{\ b :: bs\ \} \\ g \neq * \Longrightarrow \Gamma, \emptyset \vdash \langle g, \mathsf{N}(\sigma)\rangle \Downarrow \mathsf{BoolVal(true)} \\ \Gamma, \emptyset \vdash \langle e, \mathsf{N}(\sigma)\rangle \Downarrow \mathsf{BoolVal(true)}\end{array}}{\Gamma \vdash \langle (b_{while}, \mathcal{K}), \mathsf{N}(\sigma)\rangle \rightarrow_{\mathsf{AST}} \langle (b, \mathsf{blocksToCont}(bs@b_{while}, \mathcal{K})), \mathsf{N}(\sigma)\rangle}$$

$$\frac{\begin{array}{c}b_{while} = []; \texttt{while}\ (g)\ \texttt{invariant}\ e\ \{\ bs\ \} \\ g \neq * \Longrightarrow \Gamma, \emptyset \vdash \langle g, \mathsf{N}(\sigma)\rangle \Downarrow \mathsf{BoolVal(false)} \\ \Gamma, \emptyset \vdash \langle e, \mathsf{N}(\sigma)\rangle \Downarrow \mathsf{BoolVal(true)}\end{array}}{\Gamma \vdash \langle (b_{while}, \mathcal{K}), \mathsf{N}(\sigma)\rangle \rightarrow_{\mathsf{AST}} \langle ([]; \epsilon, \mathcal{K}), \mathsf{N}(\sigma)\rangle}$$

$$\frac{\begin{array}{c}b_{while} = []; \texttt{while}\ (g)\ \texttt{invariant}\ e\ \{\ bs\ \} \\ \Gamma, \emptyset \vdash \langle e, \mathsf{N}(\sigma)\rangle \Downarrow \mathsf{BoolVal(false)}\end{array}}{\Gamma \vdash \langle (b_{while}, \mathcal{K}), \mathsf{N}(\sigma)\rangle \rightarrow_{\mathsf{AST}} \langle ([]; \epsilon, \textbf{KStop}), \mathsf{F}\rangle}$$

Figure 2.28: Operational semantics for the Boogie AST. The evaluation of expressions (*e.g.* $\Gamma, \emptyset \vdash \langle g, \mathsf{N}(\sigma)\rangle \Downarrow \mathsf{BoolVal(true)}$) is defined in Figure 2.5 on page 30 and the reduction of a list of basic commands (*e.g.* $\Gamma \vdash \langle cs, \mathsf{N}(\sigma)\rangle \; [\rightarrow] \; s'$) is defined in Figure 2.7 on page 31 (we introduced both of these as part of the semantics of Boogie CFGs). The term [] denotes the empty list, and the term $b :: bs$ denotes the list whose head and tail are given by b and bs, respectively. The term $bs_1@bs_2$ denotes the list obtained by appending lists bs_1 and bs_2. The term $\mathsf{blocksToCont}(bs, \mathcal{K})$ denotes the continuation in which first the statement blocks bs are executed followed by the continuation $\mathcal{K}$ ($\mathsf{blocksToCont}(\cdot, \cdot)$ is defined by recursively applying $\textbf{KSeq}(\cdot, \cdot)$).

$$\frac{\Gamma \vdash \langle c, s \rangle \rightarrow s'}{\Gamma \vdash \langle (c :: cs; ctrl, \mathcal{K}), s \rangle \rightarrow_{\mathsf{AST2}} \langle (cs; ctrl, \mathcal{K}), s' \rangle}$$

$$\frac{\Gamma \vdash \langle ([]; ctrl, \mathcal{K}), s \rangle \rightarrow_{\mathsf{AST}} \langle \gamma', s' \rangle}{\Gamma \vdash \langle ([]; ctrl, \mathcal{K}), s \rangle \rightarrow_{\mathsf{AST2}} \langle \gamma', s' \rangle}$$

Figure 2.29: Alternative AST semantics that reduces each basic command in a separate step. The semantics defaults to the original AST semantics in the case when the list of basic commands in the active statement block is empty (*i.e.* when the next step will not execute a basic command).

To express an execution that takes 0 or more steps, we use the reflexive-transitive closure of the single step judgement. $\Gamma \vdash (\gamma, s) \rightarrow^*_{\mathsf{AST}} (\gamma, s')$ denotes the reflexive-transitive closure w.r.t. the original single step judgement defined in Figure 2.28 and $\Gamma \vdash (\gamma, s) \rightarrow^*_{\mathsf{AST2}} (\gamma, s')$ denotes the reflexive-transitive closure w.r.t. the alternative single step judgement defined in Figure 2.29.

2.8.3. AST Procedure Correctness

In Subsection 2.3.4, we provided a formal definition for procedure correctness that takes the *correctness of a procedure body* as a parameter (see Definition 2.3.1 on page 32). To define procedure correctness in the case when a procedure body is represented as an AST, we instantiate this parameter in that definition (*i.e.* in Definition 2.3.1) via the following definition that states when a *procedure body* represented by an AST is correct:

Definition 2.8.1 (Correctness of an AST body)

$$\begin{aligned}
&\mathsf{bodyCorrect}_{\mathsf{AST}}(\Gamma, body, post, \sigma) \triangleq \\
&\forall b', \mathcal{K}', s'.\ \Gamma \vdash (\mathsf{initProgPoint}(body), \mathsf{N}(\sigma)) \rightarrow^*_{\mathsf{AST}} ((b', \mathcal{K}'), s') \Rightarrow \\
&\quad s' \neq \mathsf{F} \wedge \\
&\quad \left(\begin{array}{l} (b', \mathcal{K}') = ([]; \epsilon, \textbf{KStop}) \Rightarrow \\ \forall \sigma'.\ s' = \mathsf{N}(\sigma') \Rightarrow \Gamma, \emptyset \vdash \langle post, \mathsf{N}(\sigma') \rangle \Downarrow \mathsf{BoolVal}(\mathsf{true}) \end{array} \right)
\end{aligned}$$

Here, body is a Boogie statement (i.e. a list of statement blocks) and $\mathsf{initProgPoint}(body)$ *provides the initial program point in body (i.e. a simple conversion from a list of statement blocks to a tuple where the first element is the first statement block and the second element is the continuation representing the tail of the statement block list).*

This definition is analogous to the instantiation used for the CFG (Definition 2.3.2). The program point $([]; \epsilon, \textbf{KStop})$ expresses that there is nothing left to execute, and thus the postcondition must be taken into account whenever this program point is reached.

2.9. AST-to-CFG Transformation

In this section, we briefly give an overview of our validation of Boogie's AST-to-CFG transformation that converts the AST representation of the input Boogie program (as computed by the Boogie parser) into the CFG representation. Details of the validation are presented in Aleksandar Hubanov's BSc thesis [61], which was supervised by the author of this dissertation.

[61]: Hubanov (2022), *Formally Validating the AST-to-CFG Phase of the Boogie Program Verifier*

As already discussed when showing the AST representation in Section 2.8, we support only a subset of possible AST constructs. In particular, in terms of control flow, we support while-loops but we do not support goto- and break-constructs in the AST. However, the validation of the remaining transformations that operate on the CFG still works if the CFG was obtained from goto- and break-constructs. In such a case where we do not support constructs in the AST but support all constructs in the resulting CFG, we generate certificates for the remaining transformations and combine them to get a partial result for Boogie's pipeline: we show that the validity of the VC implies the correctness of the CFG obtained from the AST-to-CFG transformation. In the case when we do support all constructs in the AST, we are able to validate the entire Boogie pipeline by additionally generating a certificate for the AST-to-CFG transformation.

The validation of the AST-to-CFG transformation works as usual via our global block theorem and local block lemma approach. For the AST-to-CFG transformation, the global block theorem relates any finite execution starting from a program point in the AST with executions starting from the corresponding basic block in the CFG. The corresponding local block lemma relates executions of the basic commands at the beginning of the program point's statement block in the AST with executions of the corresponding basic block itself.

Since we do not support gotos or breaks, we are able to prove the global block theorems by going backwards through the AST. We prove global block theorems for statement blocks with while loops by induction. The global block theorems within the loop have the corresponding induction hypothesis as a premise, analogously to how we deal with cycles in the validation of cycle elimination (Section 2.4) and the CFG optimisations (Section 2.7). One difference to these transformations is that in the AST-to-CFG transformation, we need to track only the induction hypothesis for the most inner loop that is active, instead of tracking the induction hypotheses for all active loops. This simplification is possible because we currently do not support gotos and breaks. As a result, executions exit loops only at the loop condition and thus one need not take other induction hypotheses into account even if the corresponding loop is nested within other loops. In contrast, the validation of the CFG transformations can, for instance, handle jumps out of loops from any point within the loop. To handle gotos and breaks in the AST-to-CFG validation, we would have to also support such jumps and thus would have to also track multiple induction hypotheses in global block theorems.

This concludes our discussion of the validation of Boogie's transformations. In the following sections, we will evaluate our certificate-producing implementation of Boogie, discuss related work, and finally conclude with future directions.

2.10. Implementation and Evaluation

In this section, we evaluate our certifying version of the Boogie verifier, which automatically produces Isabelle certificates that formally justify the soundness of Boogie's pipeline.

2.10.1. Implementation

We have implemented our validation tool as a new C# *project* called "ProofGeneration" as part of the existing Boogie codebase that is given by a C# *solution*.[37] We instrumented Boogie's existing codebase to call out to our C# project, which allows us to obtain information that we use to validate the transformations, and extended parts of the existing codebase to extract information more easily. Moreover, we disabled counterexample related VC features and the generation of VC axioms for any built-in types and operators that we do not support. Our validation tool currently supports the default options of Boogie (only), and one of the three possible type encodings (as discussed in Section 2.6). Our tool does not support source-level *attributes* (for instance, to selectively force procedures to be inlined).

37: A C# solution contains multiple C# projects and manages their dependencies.

Given an input file verified by Boogie, our work produces an Isabelle certificate per procedure showing that the procedure's correctness follows from the validity of the VC. As discussed in the previous sections, we cover the entire Boogie pipeline only for a subset of Boogie programs. More concretely, if the procedure is within our supported subset (see Figure 2.3 on page 25), has only features supported by our AST-to-CFG validation (*e.g.* no gotos or breaks), and has no dead variables, then our certificate covers the full Boogie pipeline. That is, in this case, the certificate shows that the correctness of the Boogie program as represented internally by Boogie's AST follows from the validity of the generated VC. If the procedure is within our subset but has features not supported by our AST-to-CFG validation, or has dead variables, then our certificate covers the entire pipeline excluding the AST-to-CFG transformation and dead variable elimination. That is, in this case, the certificate shows that the correctness of the Boogie program as represented internally by Boogie's CFG (after the elimination of dead variables) follows from the validity of the generated VC.

The generation and checking of the certificate is fully automatic, without any user input. We use a combination of custom and built-in Isabelle tactics.

2.10.2. Experimental Evaluation

We evaluated our work on two sets of benchmarks. Firstly, to evaluate the applicability of our certificate generation, we automatically collected all input files from Boogie's test suite [84], which satisfy the following con-

[84]: Boogie Developers (n.d.), *Boogie Verifier Test Suite Used For Evaluation*

Table 2.1: Selection of algorithmic examples with the lines of Boogie code (LOC), the number of Boogie procedures (#P), the time it takes for Isabelle to check the certficate in seconds (the mean of 6 runs), and the certificate size expressed as the number of non-empty lines of Isabelle. For the procedure in TuringFactorial and one of the procedures in Find the AST-to-CFG certificate was not generated since these procedures contain gotos.

Name	LOC	#P	Time [s]	Size
TuringFactorial	29	1	18.1	2361
Find	27	2	24.8	2587
DivMod	36	2	19.7	2520
Summax [82]	23	1	20.3	2547
MaxOfArray [83]	21	1	25.0	2524
SumOfArray [83]	22	1	19.9	1920
Plateau [83]	46	1	20.8	2558
WelfareCrook [83]	50	1	35.9	3241
ArrayPartitioning [83]	56	2	32.7	4304
DutchFlag [83]	75	2	33.6	4910

straints: (1) the file is successfully verified by Boogie, (2) the file has at least one procedure, and (3) the file contains only features that are supported by our certificate generation or features that can be easily desugared into our supported subset; this includes examples using procedure calls. We did this desugaring manually in all cases except for procedure calls, where we could use Boogie's call desugaring implementation to do so automatically in most cases.[38] For programs employing attributes, we checked whether the program still verifies *without* attributes, and if so we also kept these. In total, this yielded 100 programs from Boogie's test suite. Secondly, we collected a corpus of ten Boogie programs which verify interesting algorithms with nontrivial specifications: three from Boogie's test suite and seven from the literature [82, 83]. Where needed, we manually desugared usages of Boogie maps (which we do not yet support) using type declarations, functions, and axioms.

38: We had to manually desugar calls *within loops* for a technical reason regarding Boogie's implementation.

[82]: Klebanov et al. (2011), *The 1st Verified Software Competition: Experience Report*
[83]: Chen et al. (2017), *Triggerless Happy – Intermediate Verification with a First-Order Prover*

We ran our certificate-producing Boogie version on both sets of benchmarks to generate the corresponding Isabelle certificates. We then checked whether Isabelle (automatically) accepted each generated certificate. For the second set of ten Boogie programs we additionally measured the time it took for Isabelle to (automatically) accept each of the generated certificates. All experiments were run on a Lenovo T480 Ubuntu 18.04 on the Windows Subsystem for Linux with 32 GB RAM and i7-8550U 1.8 GhZ CPU (scaled up to 4 GhZ using TurboBoost). For the time measurements, we took the mean of five repetitions.

The 100 programs from Boogie's test suite contain 175 procedures.[39] For 153 procedures, our tool generates certificates for the entire Boogie pipeline. The remaining 22 procedures contain features that our tool does not yet handle in the validation of the AST-to-CFG transformation or contains dead variables. As a result, for these 22 procedures, our tool generates a certificate for the CFG representation after the dead variable elimination. Isabelle successfully checks the generated certificates for 96 of the 100 programs. The remaining 4 certificates involve special cases that we do not handle yet. For 2 of them, extending our work is straightforward: one special case includes a naming clash and the other case can be amended by using a more specific version of a helper lemma (*i.e.* by instantiating one of the universally quantified parameters in the lemma concretely). The remaining two fail because of our incomplete

39: We do not count procedures without procedure bodies. Such *abstract procedures* are trivially correct.

handling of function calls in the final generation of the VC; we have discussed the corresponding incompleteness in Subsection 2.6.4. In particular, in both cases, the main challenge is that values from certain function calls in the VC are converted from the carrier sort V (used in the VC to represent all Boogie values) to the corresponding primitive value at the VC level. Handling this is more challenging than the other two cases but is not a fundamental issue (see Subsection 2.6.4).

The corpus of 10 programs contains 14 procedures. For 12 procedures, our tool generates certificates for the entire Boogie pipeline. The remaining 2 procedures contain features that our tool does not yet handle in the validation of the AST-to-CFG transformation, because these procedures contain gotos. As a result, for these 2 procedures, our tool generates a certificate for the CFG representation after the dead variable elimination. Isabelle successfully checks all the certificates generated for all 14 procedures in the 10 programs. Table 2.1 shows the generated certificate size and the time it took for Isabelle to check their validity (the mean of five repetitions).[40] The time to generate the certificate is not included, but is negligible here. The certificate sizes are not small (ranging from 1920 to 4910 non-empty lines of Isabelle code) and the validation times (ranging from 18 to 36 seconds) are not short given the size of the programs. However, the times are acceptable since certificate generation needs to run only for the (verified) release version of the program in question or as part of continuous integration. A possible workflow could be the following. In a first step users identify suitable specifications and loop invariants for the to-be-verified program, which may also involve changes to the program itself, since the program may be incorrect. During this first step, there is no need for certificate generation, since verification either fails or the specifications and program are not yet as the user desires. In a second step, once the specification and loop invariants are finalised, and Boogie verifies the program, users can run the automatic certificate generation and can then use Isabelle to check the certificates.

40: Note that in the conference publication [59], we used a version of DivMod where we manually desugared if-then-else expressions into if-statements. Here, we use the original version, since we later added support for if-then-else expressions. Also note that the certificates in the conference publication do not cover the AST-to-CFG and the CFG optimisations, which is why the certificate sizes are larger here. Moreover, we made some general changes, which affected the certificate details after the publication.

Our evaluation demonstrates that our automatic certificate generation works on a diverse set of Boogie examples, and that Isabelle can successfully check the automatically generated certificates in acceptable times. Our evaluation does not consider Boogie programs generated by Boogie front-ends (such as Dafny and Viper), which is the main way the Boogie language is used. These front-ends currently target Boogie programs with features that go beyond our subset. Moreover, these generated programs are also significantly larger than most hand-written programs. However, we have clear ideas for how to extend our work to make automatic certificate generation applicable for such examples. In terms of larger subsets, most remaining features are straightforward to add, with some exceptions for which we have initial ideas. In terms of making the certificate generation scale to large programs, we are aware of one bottleneck for which we will discuss a concrete solution in Subsection 2.12.1 that should eliminate this bottleneck. Finally, for the incompletenesses in the certificate generation, which we have noticed in our evaluation, we have clear solutions. We will discuss directions for adding support for front-end-generated Boogie programs in Subsection 2.12.1.

2.10.3. Trusted Components

Our certificate-producing version of Boogie greatly reduces the parts of the Boogie verifier implementation that must be trusted. In particular, if Isabelle successfully checks the generated certificate, then Boogie's transformations from the AST representation (or CFG representation if the program has breaks or gotos) to the VC need not be trusted.

However, there are still various parts that must be trusted in order to conclude that a certificate successfully checked by Isabelle actually implies that the VC produced by Boogie implies the correctness of the input Boogie program, which include:

- the soundness of the Boogie parser that translates a source program represented in text into Boogie's internal AST representation
- our deep embedding of the Boogie AST representation in Isabelle must reflect the input program; this includes our formal Boogie semantics
- our shallow embedding of the VC in Isabelle must reflect the verification condition that Boogie sends to the SMT solver
- the soundness of Isabelle

[85]: Jourdan et al. (2012), *Validating LR(1) Parsers*

There is existing work that provides a formal technique to increase the trustworthiness of parsers [85]. The semantics of the embedded Boogie program is a fundamental trust assumption, which we cannot fully eliminate, since soundness is defined w.r.t. this semantics. However, we could increase the confidence that our formalised semantics matches the intended semantics by proving the correctness or incorrectness of concrete example Boogie programs in Isabelle w.r.t. the formalised semantics. We could obtain the intended result (*i.e.* correctness or incorrectness) from the Boogie test suite, which makes explicit which programs in the test suite are supposed to be correct and which programs are supposed to be incorrect. Similarly our shallow embedding of the VC is also a fundamental trust assumption, for which we could similarly increase our confidence that this embedding indeed reflects the intended semantics. Finally, Isabelle and other interactive theorem provers (ITPs) have an isolated kernel that must be trusted. Since Isabelle is a mature tool developed with the purpose of establishing trustworthy guarantees, which has many users, its kernel (and thus Isabelle itself) is considered by the research community to be extremely trustworthy. Nevertheless, there is existing work that shows how to formally prove the soundness of an ITP [86].

[86]: Abrahamsson et al. (2022), *Candle: A Verified Implementation of HOL Light*

The soundness of the parser is not relevant for Boogie front-ends if one directly shows that the correctness of the front-end program is implied by the correctness of the corresponding Boogie encoding as represented internally by Boogie's AST (as we will do in Chapter 3 for Viper). In some cases, the parser is not even used by a front-end, since front-ends may directly construct Boogie programs using Boogie's internal AST representation; for example, Dafny does so by using Boogie as a C# library.

Finally, to conclude that the Boogie program is correct for a successful verification result (without assuming the validity of the VC), one must additionally trust that the VC generated by Boogie is indeed valid. To avoid trusting this component, one could build on existing work that makes SMT solvers certificate-producing [8–10] to automatically prove in Isabelle that the generated VC is valid.

[8]: Böhme et al. (2010), *Fast LCF-Style Proof Reconstruction for Z3*
[9]: Ekici et al. (2017), *SMTCoq: A Plug-In for Integrating SMT Solvers into Coq*
[10]: Fleury et al. (2019), *Reconstructing veriT Proofs in Isabelle/HOL*

Boogie's type checker

We need not trust Boogie's type checker, because our certificates do not explicitly assume that the input program is well-typed. In particular, the trust assumption on our deep embedding of Boogie's AST representation in Isabelle and on our Boogie semantics captures whether our deep embedding accurately models the input program. However, our semantics accurately models only Boogie programs that are well-typed. For instance, an assignment $x := e$ reduces only if e evaluates to a value whose type matches the type declared for variable x. So, to weaken our trust assumption on our Boogie semantics, we could increase the trustworthiness of our Boogie semantics by: (1) proving a type soundness result for our operational semantics, which states that if the input program is well-typed, then executions cannot get *stuck*, and (2) proving that the input program is well-typed. For (2), we could either trust Boogie's type checker or automatically generate a certificate that proves well-typedness of a program.

Combining certificates for front-end translations with our generated certificates

If one establishes a certificate showing the soundness of a front-end translation *into* Boogie and then combines this certificate with the certificate generated by our instrumented Boogie version, then one need not trust the Isabelle embedding of the Boogie AST representation and the Boogie semantics. The reason is that in this case the combined certificate establishes that the validity of the VC generated by Boogie implies the correctness of the front-end program. Since the validity of the VC and the correctness of the front-end program are independent from the corresponding Boogie program, one need not trust the Boogie semantics. In this case, the Boogie program just serves as an intermediate point that connects two different certificates.

2.11. Related Work

Several works explore the validation of SMT-based program verifiers for a subset of the language supported by the verifier. Garchery [35] validates VC rewritings in the existing Why3 implementation [17]. Unlike our work, they do not connect VCs with programs and do not handle the erasure of polymorphic types. Strub et al. [56] develop a self-certification approach and apply it to the dependently-typed F* language. They generate a Coq certificate for a core F* type checker written in F*. Like us, they do not certify the validity of conditions encoded into an SMT query, but they do not consider program-to-program transformations such as the ones that we validate. Aguirre [87] shows how one can map proofs for F*'s logical encoding (which is given to an SMT solver) back to a proof of the corresponding F* program. They prove a once-and-for-all result, but since their once-and-for-all proof is constructive, their approach could be ported to a per-run validation approach using the certificate-

[35]: Garchery (2021), *A Framework for Proof-carrying Logical Transformations*
[17]: Filliâtre et al. (2013), *Why3 — Where Programs Meet Provers*
[56]: Strub et al. (2012), *Self-certification: bootstrapping certified typecheckers in F* with Coq*
[87]: Aguirre (2016), *Towards a Provably Correct Encoding from F* to SMT*

[88]: Barrett et al. (2015), *Proofs in Satisfiability Modulo Theories*

producing capability of SMT solvers [88]. Porting the approach would require extending their approach to handle classical proofs (as produced by certificate-producing SMT solvers) instead of constructive proofs of logical formulas. There are also existing works that validate existing program verifier implementations based on symbolic execution, which require a fundamentally different validation approach, since symbolic execution works very differently compared to Boogie's VC generation. Lin et al. [48] validate such verifiers obtained via the K framework, and Wils and Jacobs [46] validate VeriFast [89].

[48]: Lin et al. (2023), *Generating Proof Certificates for a Language-Agnostic Deductive Program Verifier*

[46]: Wils et al. (2023), *Certifying C program correctness with respect to CH2O with VeriFast*

[89]: Jacobs et al. (2011), *VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java*

There is work on implementing VC generators in an interactive theorem prover and then proving the implementation sound once and for all, although none of the proven tools are used in practice. Homeier and Martin [32] prove a VC generator sound in HOL for an executable programming language and a simpler VC generation technique than Boogie's. They run their VC generator within HOL, which produces a set of verification conditions that a user needs to prove manually in HOL. Herms et al. [90] prove a VC generator inspired by Why3 sound in Coq. However, some more-challenging aspects of Why3's VC transformation and polymorphic type system are not handled. They extract their implementation to OCaml and combine it with existing solvers, which results in a fully automated executable tool. Cohen and Johnson-Freyd [36] implement and prove sound two VC rewritings performed by Why3 in Coq to demonstrate their novel Why3 mechanisation. They do not consider VC generation itself. Blatter et al. [34] prove a VC generator sound in Coq, which verifies relational properties for programs with pointers. Vogels et al. [31] prove a toolchain for a Boogie-like language sound in Coq, including an assignment elimination transformation and VC generation. However, the language is quite limited: without unstructured control flow,[41] loops (*i.e.* no need for eliminating cycles), functions, or polymorphism (*i.e.* no type encoding). Their (once-and-for-all) proof of the soundness of assignment elimination is set up differently to our (per-run generated) proof. Their implementation's VC generation is based on an efficient weakest precondition for structured control flow [22], which is different from an efficient weakest precondition for unstructured control flow used in Boogie [23].

[32]: Homeier et al. (1995), *A Mechanically Verified Verification Condition Generator*

[90]: Herms et al. (2012), *A Certified Multi-prover Verification Condition Generator*

[36]: Cohen et al. (2024), *A Formalization of Core Why3 in Coq*

[34]: Blatter et al. (2022), *Certified Verification of Relational Properties*

[31]: Vogels et al. (2010), *A machine-checked soundness proof for an efficient verification condition generator*

41: Recall that we do support unstructured control flow in the CFG representation (but not in the AST representation).

[22]: Leino (2005), *Efficient weakest preconditions*

[23]: Barnett et al. (2005), *Weakest-precondition of unstructured programs*

There are also other kinds of once-and-for-all proved verifiers. The Verasco static analyzer [91] is proved in Coq. Verasco supports a realistic subset of C, but its performance is not yet on par with unverified, industrial analyzers. Doenges et al. [92] prove the soundness of an algorithm in Coq, which verifies the equivalence of protocol parsers. Their algorithm is expressed as an inductive relation. Running the algorithm corresponds to executing a tactic within Coq that automatically finds a derivation using the rules of the relation. For some of the conditions that arise during the proof search, the tactic applies a once-and-for-all proved translation to an SMT formula, which is then handed to an SMT solver; as in our case, the SMT solver is currently trusted.

[91]: Jourdan et al. (2015), *A formally-verified C static analyzer*

[92]: Doenges et al. (2022), *Leapfrog: certified equivalence for protocol parsers*

Per-run validation has been explored in settings different from program verification. Alkassar et al. [93] adjust graph algorithms to produce witnesses that can be then used by verified validators to check whether the result is correct. In the context of compiler soundness, many validation techniques express a per-run validator in Coq, prove it sound once and for all, and then extract executable code (the extraction must typically

[93]: Alkassar et al. (2014), *A Framework for the Verification of Certifying Computations*

be trusted) [69, 94, 95]. In the verified CompCert compiler [68], such validators have been used in combination with the once-and-for-all approach. Validators are used for transformations that can be more easily validated than proved sound once and for all. One such example related to our certification of assignment elimination is the validation of CompCert's SSA transformation [69], dealing also with versioned variables in the target (but not with `assume` commands that prune executions). In contrast to our work, they require an explicit notion of CFG domination and they do not use a global versioning scheme to efficiently check that two parts of the CFG constrain disjoint versions. Our global versioning idea is similar to a technique used for the validation of a dominator relation in a CFG [96], which assigns intervals to basic blocks (as opposed to assigning versions to variables) to efficiently determine whether a block dominates another one. The validation of the Cogent compiler [39] follows a similar approach to ours in that it generates certificates in Isabelle.

[69]: Barthe et al. (2014), *Formal Verification of an SSA-Based Middle-End for CompCert*
[94]: Tristan et al. (2008), *Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations*
[95]: Tristan et al. (2010), *A simple, verified validator for software pipelining*
[68]: Leroy (2006), *Formal certification of a compiler back-end or: programming a compiler with a proof assistant*
[69]: Barthe et al. (2014), *Formal Verification of an SSA-Based Middle-End for CompCert*
[96]: Blazy et al. (2015), *Validating Dominator Trees for a Fast, Verified Dominance Test*
[39]: Rizkallah et al. (2016), *A Framework for the Automatic Formal Verification of Refinement from Cogent to C*

It would be interesting to explore, in our setting, the trade-offs of using a per-run validator that is proved once and for all (as used in CompCert) compared to our approach of generating certificates in Isabelle. As a first step toward this direction, one could attempt such an alternate approach solely for validating our local block lemmas. Our current approaches for automatically proving local block lemmas are largely syntax-directed: we automatically prove relational judgements by applying rules that match syntactically on the goal to be proved (see Subsection 2.4.4 on page 42, Subsection 2.5.3 on page 54, and Subsection 2.6.4 on page 68). One could define a validator as a Boolean function in Isabelle that essentially mimics our syntax-directed proof search and returns true if the proof search succeeds.

Finally, various works present a semantics for a Boogie subset, but they either handle a smaller subset than our formalised subset, or do not formalise all aspects of their presented semantics. Leino [1] describes the semantics for the subset covered in this dissertation on paper. In their work, various parts are not given a formal semantics, but rather discussed at a higher level. Vogels et al. [31] mechanise a small subset of Boogie in Coq, but do not handle unstructured control flow, loops, and global declarations. Moreover, they represent Boogie expressions semantically (*i.e.* as functions from Boogie states to values), instead of syntactically in our case. Thus, they do not, for instance, give a semantics to the quantification over types. Grigore [97] formalises a similar subset on paper as Vogels et al. [31] and they also model expressions semantically in their semantics. In contrast to Vogels et al. [31], they include unstructured control flow.

[1]: Leino (2008), *This is Boogie 2*
[31]: Vogels et al. (2010), *A machine-checked soundness proof for an efficient verification condition generator*
[97]: Grigore (2012), *The Design and Algorithms of a Verification Condition Generator*

2.12. Future Work

In this section, we discuss some avenues for future work.

2.12.1. Support for Front-End-Generated Boogie Programs

Our work targets an important core subset of Boogie that captures many of the challenges that Boogie's implementation faces. However, our supported subset is not yet large enough to support Boogie programs generated by Boogie front-ends, which is the main way Boogie is used. Thus, one important direction for future work is to extend our work such that it can be applied to programs generated by Boogie front-ends. There are two subdirections to achieve this goal: (1) increase the Boogie subset for which certificates are generated, and (2) ensure that Isabelle is able to check the generated certificates also for Boogie programs generated by Boogie front-ends.

For subdirection (1), features that are currently unsupported and that are used by front-ends include: more built-in types (such as maps, bitvectors, reals), procedure calls, conditions on variable declarations (called *where-clauses* in Boogie) that ensure the specified conditions hold whenever the variable is havocked, gotos and breaks. Some of these can be expressed via our subset and thus are not challenging (such as procedure calls and where-clauses). Two of the most challenging features are maps and gotos. We will discuss maps below in Subsection 2.12.2. For gotos, we already support the CFG transformations applied on the CFG obtained from an AST with gotos (and breaks). The challenge is to add goto support for the AST-to-CFG transformation. One challenge here is to do so efficiently with Boogie's internal AST representation, which is not straightforward since the standard small-step goto semantics for such an AST computes the continuation by traversing the entire program.

For subdirection (2), there are two key points. First, the certificates for the final generation of the VC must be made more robust to avoid, for example, the incompleteness with function calls and conversions observed in Section 2.10, since such instances also arise in front-end-generated Boogie programs. As we have discussed in Subsection 2.6.4, improving the automation to deal with the function call incompleteness should be straightforward. Second, we must make sure that the generated certificates are sufficiently optimised such that Isabelle is able to check these certificates in reasonable time also for large programs. There is one bottleneck currently that shows up for large programs, but which we should be able to optimise away using existing techniques, as we discuss next.

Proofs in our certificate often need to show that a Boogie variable is declared with a concrete type. In the Isabelle embedding of the Boogie program, the declared variables are represented via a list of tuples, each tuple consisting of a variable name and its declared type. We prove a lemma once for each declared variable that states what its declared type is (*i.e.* essentially creating a lookup table mapping Boogie variables to their types) and then use this lemma in proofs. The problem is that each proof currently results in Isabelle performing a linear search through the list of declared variables (until the first matching tuple is reached), which means the entire proof effort in Isabelle for these lemmas is quadratic in the number of Boogie variables (since there is one lemma per variable). This can lead to slow proofs, since front-end-generated Boogie programs can have a large number of variables (moreover, assignment elimination creates a fresh variable for every variable update). This process can be optimised using existing techniques. One approach would be to represent the declared variables in a binary search tree, which would make the proof effort for a single lemma logarithmic instead of linear in the size of the declared variables. The Isabelle seL4 kernel verification code already provides a generic library for working with such a binary search tree approach [98]. We have performed some initial experiments in a simplified setting (incorporating such a library into our certificate generation architecture will take more work), which confirm that such a binary search tree approach significantly speeds up the time for proving the lemmas.

[98]: seL4 Developers (n.d.), *Efficient lookup table creation in Isabelle*

2.12.2. Support for Maps

Boogie supports two kinds of Boogie maps: *non-polymorphic* and *polymorphic* maps. Both maps are used in practice: the former is used in many usages of Boogie, while the latter is used mainly by Boogie front-ends modelling a heap. Thus, adding support for both kinds is important. However, providing a formal model is challenging in both cases as we will discuss next.

Non-polymorphic maps

A non-polymorphic map is a standard total map, which is specified via domain types $D_1, D_2, ..., D_n$ and a range type R (the corresponding map type is written as `[D1,D2,...,Dn]R` in Boogie) and maps a tuple of values $(v_1, v_2, ..., v_n)$ where v_i is of type D_i to a value of the range type R.

It is not clear how to formalise non-polymorphic maps in Isabelle. A naive approach would be to extend the definition of Boogie values as follows:

$$'a\ \mathsf{val} \triangleq \mathsf{IntVal}(i) \mid \mathsf{BoolVal}(b) \mid \mathsf{AbsVal}('a) \mid \mathsf{MapVal}('a\ \mathsf{val\ list} \Rightarrow\ 'a\ \mathsf{val})$$

This extended definition contains an additional constructor `MapVal` for the map values, which are represented by functions from lists of values to values. This is an ill-formed definition: There exists no interpretation of $'a$ val that satisfies the equation, since there will always be more functions from type $'a$ val list $\Rightarrow$ $'a$ val than values of type $'a$ val (if the interpretation of $'a$ val is non-empty, which it must be due to the other constructors).

So, to support maps one must first find a way to model them formally. In many practical cases, only maps of finite size are needed. Restricting maps to have finite size would make the above definition well-formed (*i.e.* replace $'a$ val list $\Rightarrow$ $'a$ val by the type of maps of finite size from $'a$ val to $'a$ val). The definition remains well-formed if one restricts maps to be of countable size. In both of these cases (finite and countable), values not in the domain could be mapped to some default value. However, it turns out that there are cases where maps of uncountable size might be required. For instance, the Boogie front-end Dafny supports the `imap` type, which models maps of infinite size. These maps are encoded into Boogie maps, which have the same domain and range types as the `imap`. The domain of an `imap` does not have any restriction and can thus have uncountable size (Dafny accepts reals as the domain). We do not know to what extent such maps of uncountable size in Dafny are used, but if they are used, then one needs a more general solution.

Polymorphic maps

Polymorphic maps are like non-polymorphic maps except that they additionally contain type parameters that the domain and range types depend on. For instance, the standard polymorphic map to model a heap has the polymorphic map type `<T>[ref, field T]T`: a total map storing, for *any* type `T`, values of type `T` given (as key) a reference and field with type argument `T`. (Note that `ref` and `field` are type constructors.) The advantage of using a polymorphic map in this case is that a single map can be used such that a lookup with a field f evaluates to a value of the type associated with f. With non-polymorphic maps, one would need to either model a single universe type that captures all possible values and use coercion functions between the universe type and the original types, or one would need to track multiple non-polymorphic maps.

To our knowledge, there exists no formal model for Boogie's polymorphic maps. Providing a general model is challenging: in particular, Boogie's polymorphic maps are *impredicative* in general: a map m of type `<T>[T]T'` permits *any* value as a key, including the map m itself! However, front-ends use polymorphic maps typically in restricted ways (*e.g.* without using an impredicative type). So, a simpler solution is to provide a formal model that explains concrete instances of polymorphic maps that are used by front-ends. In Chapter 3, we will show an approach to do so for the polymorphic map type used by Viper for the heap. Instead of adding the polymorphic map type as a first-class citizen to our Boogie formalisation, we model the type via type constructors and polymorphic functions, which allows us to reuse our existing Boogie formalisation. This requires representing map lookups and updates differently in the Boogie program (via calls to the introduced polymorphic functions). (This approach could also be taken for non-polymorphic map types, but since many different variations are used compared to a small finite and statically fixed number of polymorphic map types for a front-end, this approach does not scale to non-polymorphic map types.) Finally, note that the Boogie implementation performs a similar desugaring step internally for polymorphic maps.

2.12.3. Monomorphisation

Boogie supports three type encodings to desugar Boogie's type system. Our certificate-producing tool currently supports only one of the encodings as discussed in Section 2.6, which we refer to as the *predicate encoding* in this subsection. In particular, we do not support the *monomorphisation* encoding, which (1) eliminates the polymorphism of a Boogie program in a separate transformation by considering each *relevant* instantiation of the type parameters (*i.e.* each instantiation that appears in the program), and (2) desugars type constructors with multiple arguments into nullary type constructors (one for each relevant instantiation). When we finished our tool, the monomorphisation encoding did not support the quantification of types or polymorphic maps. As a result, Boogie by default used the predicate encoding with these features. More recently, Boogie developers extended monomorphisation to the quantification of types and polymorphic maps [99], and it is now the default encoding for programs with these features. So, it would be useful to add support for

[99]: Qadeer (2022), *Monomorphization of polymorphic maps and binders*

```
type option _;
function some<T>(i: T) : option T;
function f<T>(i: T) : bool;
axiom (forall <T> :: (forall i : T :: f(i) == f(some(i))));

procedure p(x: int) {
  assume f(x);
  ...
}
```

Figure 2.30: Boogie program that cannot be monomorphised by the existing Boogie verifier, since infinitely many instantiations would be taken into account. The code after the `assume` command in procedure `p` is not provided explicitly; this `assume` command is sufficient to trigger infinitely many instantiations.

the monomorphisation encoding given that it is now the default encoding for most relevant Boogie programs. One challenge in formally validating monomorphisation is justifying the monomorphisation of universal type quantification, since monomorphisation essentially quantifies over only a subset of all possible types.

While monomorphisation is the default encoding, the predicate encoding still has advantages compared to monomorphisation in some cases. One advantage is that for certain programs, the predicate encoding currently performs better. For instance, the predicate encoding performs better for Boogie programs generated by the existing Viper-to-Boogie implementation. We have not investigated the details, but one reason seems to be that monomorphisation must consider very many instantiations for the Boogie programs generated by the Viper-to-Boogie implementation. One could look into ways of making the generated Boogie programs more suitable for monomorphisation.

A second advantage is that the predicate encoding is able to handle programs that cannot be monomorphised. For instance, Boogie cannot monomorphise the program shown in Figure 2.30, but Boogie is able to successfully apply the predicate encoding to this program. The reason Boogie cannot monomorphise this program is that Boogie's monomorphisation currently would take *infinitely* many instantiations into account for the type parameter `T` in the definition of function `f`. In particular, the `assume` command in procedure `p` instantiates `T` with `int`, which triggers infinitely many instantiations of `T` via the axiom in the program. The reason is that for any considered instantiation τ, the axiom also contains the instantiation `option` τ (because of the expression `f(some(i))`). Thus, Boogie's monomorphisation would consider instantiations `int`, `option int`, `option (option int)`, and so on.

Supporting the development of monomorphisation

When Boogie developers were working on extending their monomorphisation approach to polymorphic maps and type quantification in 2022 (which was eventually merged [99]), we were able to support them by formally explaining what a feasible formal semantics for these features is, using our work as a justification for this semantics. For type quantification, we define a formal semantics in Section 2.3 on page 24 and justify this semantics in two ways: (1) we demonstrate that Boogie's generated verification condition respects this semantics, and (2) we demonstrate that this semantics can be used to justify existing front-end translations into Boogie (as we show via our work in Chapter 3). Our work does not provide a formal semantics for general polymorphic maps, but in Chapter 3 we demonstrate how to formally capture instances of polymorphic maps used in practice (*e.g.* to represent heaps), which helped gain a deeper understanding for their intended meaning.

This concludes this chapter on the formal validation of the existing Boogie verifier implementation. Our work makes the existing implementation certificate-producing, thus significantly increasing the implementation's trustworthiness. The next chapter will present the formal validation of a front-end translation into Boogie. The corresponding soundness results are expressed w.r.t. the formal Boogie semantics developed in this chapter, thus showing that our formalised Boogie semantics can be used to justify the soundness of a front-end translation.

Formally Validating Translations into an Intermediate Verification Language 3.

3.1. Introduction

As discussed in Section 1.1 on page 3, many translational program verifiers translate the input program into an intermediate verification language (IVL); we call this a *front-end translation*. An IVL comes with its own *back-end verifier* that ultimately reduces IVL programs to logical formulas (such as the Boogie verifier, which is the focus of Chapter 2). There are many examples of practical front-end translations; *e.g.* Corral [15], Dafny [3], SMACK [16], SYMDIFF [71] and Viper [7] translate to the imperative Boogie IVL [1]. Creusot [2] and Frama-C [4] translate to the functional Why3 IVL [17]. Multiple layers of front-end translations and IVLs can also be *composed* (*e.g.* Prusti [19] builds on Viper as an IVL).

[15]: Lal et al. (2014), *Powering the static driver verifier using corral*

[3]: Leino (2010), *Dafny: An Automatic Program Verifier for Functional Correctness*

[16]: Carter et al. (2016), *SMACK software verification toolchain*

[71]: Lahiri et al. (2012), *SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs*

[7]: Müller et al. (2016), *Viper: A Verification Infrastructure for Permission-Based Reasoning*

[1]: Leino (2008), *This is Boogie 2*

[2]: Denis et al. (2022), *Creusot: A Foundry for the Deductive Verification of Rust Programs*

[4]: Kirchner et al. (2015), *Frama-C: A software analysis perspective*

[17]: Filliâtre et al. (2013), *Why3 — Where Programs Meet Provers*

[19]: Astrauskas et al. (2019), *Leveraging Rust Types for Modular Specification and Verification*

To ensure that successful verification indeed implies that the input program conforms to its specification, translational program verifiers that apply a front-end translation must meet two *soundness conditions*: (1) *front-end soundness*: the *translation* into the IVL is faithful, *i.e.* correctness of the produced IVL program implies correctness of the input program, and (2) *IVL back-end soundness*: if the IVL back-end verifier reports success, then the IVL program is correct. In this chapter, we focus on establishing front-end soundness for existing verifier implementations used in practice. IVL back-end soundness is an orthogonal concern. Our results on front-end soundness can be combined with work on establishing IVL back-end soundness to obtain end-to-end guarantees; this includes our own IVL back-end verifier results developed for Boogie in Chapter 2, and other works (including the validation of SMT solvers) [8–10, 35].

[8]: Böhme et al. (2010), *Fast LCF-Style Proof Reconstruction for Z3*
[9]: Ekici et al. (2017), *SMTCoq: A Plug-In for Integrating SMT Solvers into Coq*
[10]: Fleury et al. (2019), *Reconstructing veriT Proofs in Isabelle/HOL*
[35]: Garchery (2021), *A Framework for Proof-carrying Logical Transformations*

Existing work on ensuring front-end soundness is based on idealised implementations that are formalised on paper or in an interactive theorem prover. These idealised implementations typically do not consider optimisations performed by more practical verifier implementations. As discussed in Chapter 1, there is a large gap between these implementations and existing verifier implementations used in practice. In this chapter, we bridge this gap for the first time, developing an approach to formally validate the front-end soundness of translations used in *existing, practical* verifier implementations.

Proving front-end soundness once and for all for a realistic verifier implementation is practically infeasible, since such implementations of front-end translations are large (*e.g.* 17.2 KLOC and 8.5 KLOC for the Dafny-to-Boogie and Viper-to-Boogie front-ends, respectively) and are typically written in languages that lack a full formalisation (*e.g.* C# for Dafny-to-Boogie and Scala for Viper-to-Boogie). Instead, we develop a formal translation validation approach that, given a formal semantics for the input language and IVL, *automatically* generates a formal certificate on every run of the verifier via an instrumentation of the existing implementation. Our certificates are expressed in the Isabelle theorem prover [12], and thus provide formal and trustworthy guarantees, effectively removing the (substantial) front-end translation from the trusted code base of the verifier. Our approach ensures that

[12]: Nipkow et al. (2002), *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*

the generated certificates contain sufficient information for Isabelle to automatically check them. Ensuring the automatic checking of certificates is crucial, since their size and complexity make manually checking them practically infeasible.

3.1.1. Challenges

Formally validating front-end translations is challenging for three main reasons:

1. Semantic gap: There is a large semantic gap between a front-end language and an IVL, which is due to large differences in the state model, execution model, and program logics used to reason about programs. For instance, states in most front-end languages have a heap, while Boogie and Why3 do not. Moreover, the evaluation of expressions and execution of statements differ significantly between a front-end language and an IVL. For instance, Viper heap accesses are partial operations that must be guarded by semantic conditions ultimately checked by verification, while Boogie and Why3 use syntactic checks to guard state accesses such as disallowing global variables in Boogie axioms and restricting aliasing between mutable variables in Why3. Moreover, the execution of certain Viper statements is complex since the execution must take complex assertions into account, while the execution of Boogie statements is straightforward. Finally, front-ends use complex program logics, such as dynamic frames [100] in Dafny, a flavour of separation logic [33, 101] in Viper, and prophetic reasoning in Creusot [2], whereas Boogie and Why3 do not have built-in support for such logics. To bridge the semantic gap, front-ends translate input programs into a complex combination of low-level operations (*e.g.* nondeterministic assignments, `assume` and `assert` commands) and background logical axiomatisations of input language concepts (*e.g.* axiomatising the consistency of a front-end state). Formal validation needs to precisely account for the combination of these ingredients, while allowing the separation of translation aspects for the sake of modularity and maintainability.

[100]: Kassios (2006), *Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions*

[33]: Smans et al. (2012), *Implicit Dynamic Frames*

[101]: Parkinson et al. (2012), *The Relationship Between Separation Logic and Implicit Dynamic Frames*

[2]: Denis et al. (2022), *Creusot: A Foundry for the Deductive Verification of Rust Programs*

2. Diverse translations: Practical front-end translations are *diverse* in the sense that they use multiple alternative translations for the same feature, *e.g.* more efficient alternative translations that are sound only in certain cases. These translations also evolve frequently over time, as new techniques and features are developed or optimised; to be practically useful, a formal approach to validation should provide means of minimising the impact of the exchange of one translation for another.

3. Non-locality: The soundness of practical translations of a fragment of the input program may depend on several checks that are performed at different places in the IVL program. For instance, the translation of a procedure call might be sound only because well-formedness of the procedure specification has been checked elsewhere in the generated IVL code. Such non-local checks are commonly used to speed up verification, for instance, to check well-formedness conditions once and for all rather than each time a specification is used. However, they complicate the soundness argument for the translation, which needs to somehow track the dependencies on properties checked elsewhere.

3.1.2. This Chapter

We present the first approach for enabling automatic formal validation for existing implementations of the front-end translations employed in many practical program verifiers. This validation guarantees front-end soundness and, thus, makes automated program verifiers substantially more trustworthy.

The core of our approach is a general methodology for generating *forward simulations* [74] between the statements of the input and the IVL program in a modular way. Our methodology provides solutions to the three challenges above. It (1) bridges the semantic gap with a novel approach by which the simulation proof is split into smaller simulations, (2) supports diverse translations by parameterising simulations along multiple dimensions (*e.g.* parameters for the state relation, the to-be-simulated effect, and the IVL code itself), and (3) handles non-locality by systematically and formally tracking dependencies during a simulation proof.

[74]: Lynch et al. (1995), *Forward and Backward Simulations: I. Untimed Systems*

For concreteness, we present our methodology for the translation from a core fragment of Viper to Boogie, as implemented in an existing and actively-used verification tool [102]. This translation is significant because it exhibits all of the challenges discussed above and because both Viper and Boogie are widely used. While our methodology is phrased in terms of Viper and Boogie, we have designed our approach, which solves the key challenges above, to generalise to other front-end translations (*e.g.* the Dafny-to-Boogie translation).

[102]: Viper Developers (2024), *Viper-to-Boogie implementation (https://github.com/viperproject/carbon)*

We have applied our methodology via an instrumentation of the existing Viper-to-Boogie translation such that on every run of the translation, an Isabelle certificate establishing front-end soundness is automatically produced (for a subset of Viper programs). The produced certificates can be automatically checked by Isabelle. This certificate-producing support for the existing Viper-to-Boogie translation is an important result, since Viper is used by many program verifiers for mainstream programming languages. Such verifiers include Gobra (Go) [5], Prusti (Rust) [19], Nagini (Python) [18], and VerCors (Java) [6]. These verifiers use Viper as an IVL: they translate the input program into Viper via a front-end translation. Thus, the soundness of each of these verifiers relies on the Viper verifiers being sound. A key feature of Viper is that Viper has a built-in heap and supports separation logic [20] reasoning about heap-manipulating programs. As a result, for separation logic verifiers, it is easier to implement a translation into Viper than using another IVL that does not have support for separation logic reasoning. While Viper's main use case is as an IVL, we will treat Viper as a front-end language that is translated to Boogie in this chapter.

[5]: Wolf et al. (2021), *Gobra: Modular Specification and Verification of Go Programs*

[19]: Astrauskas et al. (2019), *Leveraging Rust Types for Modular Specification and Verification*

[18]: Eilers et al. (2018), *Nagini: A Static Verifier for Python*

[6]: Blom et al. (2017), *The VerCors Tool Set: Verification of Parallel and Concurrent Software*

[20]: Reynolds (2002), *Separation logic: A logic for shared mutable data structures*

We have applied our methodology to the *existing* Viper-to-Boogie translation for a core *subset* of Viper. However, we have taken care throughout this work to develop general approaches that could be used for different Viper-to-Boogie translations and that could be extended to larger subsets of Viper. First, this is exemplified by the fact that our methodology itself is general and can be applied to front-end translations for source languages other than Viper. Second, the concrete formalisation of our methodology, which is expressed in terms of Viper (as source language) and Boogie (as target IVL), captures many more possible translations than just the existing Viper-to-Boogie translation due to our approach being generic along multiple dimensions (*e.g.* our approach is generic in the state relation that connects source program states with corresponding IVL program states, and is also generic in the precise IVL statements used to capture a source construct). Third, while we generate certificates for a core subset of Viper, we formalised a larger Viper subset and took this larger subset into account for the metatheory involved in the application of our methodology, since there are features outside of our core subset, which significantly impact the formalisation (such as the Viper *permission introspection* feature). Moreover, we also took care to make our approach extensible to Viper features that we did not formalise. Thus, we ensure that extensions to larger Viper subsets will not require substantially changing the existing results. Fourth, we develop approaches that are designed to scale to more complex translations and larger language subsets, such as finding systematic ways of using a single approach to deal with different aspects of the language, breaking down the original problem into small subproblems, and proving generic results that can be reused for different translations without much extra work.

One challenge specific to making our approaches extensible to larger Viper subsets than our core subset (for which we support certificates) was that we could not reuse an existing formal semantics of Viper. Previous attempts at formalising Viper features in such larger Viper subsets did not model certain features accurately, and did not consider the implications of combining certain features. Two examples for such features are permission introspection and *Viper predicates*. So, part of our work on ensuring extensibility was investigating the intended semantics of such features. We included permission introspection as part of our formalisation, which also had a significant impact on our formalisation of features in our core subset. Moreover, we discovered novel insights on the semantics of Viper predicates; we did not formalise Viper predicates, but we included constructs in our formalisation to ease an extension to predicates as a consequence of our insights. In particular, we discovered that the combination of Viper *unfolding expressions* (a Viper feature used with predicates) and permission introspection has an unclear semantics. This lack of clarity likely does not impact practical Viper programs, but resolving the meaning of this combination is important to clarify what both of these features mean and also will help future use cases that may rely on this combination. We discuss another novel insight, which fundamentally impacts how to formalize Viper predicates, as part of our discussion on future work (Subsection 3.9.1 on page 195).

One reason why we were able to identify such novel insights is because our goal was to *formally* capture the connection of a verifier *implementation* that is *used in practice* with a Viper semantics. For certain features outside of our core subset (such as permission introspection and predicates), we analysed how the Viper-to-Boogie implementation was handling these features, and explored whether our formalised semantics and validation approach would be easily extensible to such features.

The work in this chapter had an impact on the existing Viper-to-Boogie implementation. We discuss some of this impact in Section 3.8. In particular, we improved various aspects of the existing implementation (*e.g.* generating better error messages and simplifying the implementation), and we discovered two soundness bugs, one of which we fixed. The other bug is outside of our formalised subset (involving predicates), which we have not yet fixed because fixing the bug requires answering design questions for Viper. This bug arises in a corner case that likely does not impact practical Viper programs, but fixing the bug is still important and will improve Viper as a language due to the design questions that arose as a result of this bug.

Contributions

We make the following technical contributions:

- We develop a general methodology for the automatic validation of front-end translations based on forward simulation certificates. We present this methodology for the translation from Viper to Boogie. As a foundation for the certificates, we formalise a semantics for a core subset of Viper in Isabelle and use our Isabelle formalisation for Boogie presented in Chapter 2.
- We instrument the existing Viper-to-Boogie implementation such that on every run, for a subset of Viper, it automatically generates an Isabelle certificate justifying the soundness of the translation. These generated certificates can be checked automatically in Isabelle, which ensures front-end soundness of the Viper verifier.
- Our evaluation on a diverse set of Viper programs demonstrates our approach's effectiveness: we were able to generate certificates and check them in Isabelle fully automatically in all cases.
- As part of justifying the axioms used in Boogie programs, we provide the first approach to formally deal with a restricted version of Boogie's (impredicatively-)*polymorphic maps* [67].

[67]: Leino et al. (2010), *A Polymorphic Intermediate Verification Language: Design and Logical Encoding*

Outline

Section 3.2 provides the necessary background on Viper, and introduces our novel semantics for a Viper subset. Section 3.3 illustrates the general challenges for justifying front-end translations presented above on the existing Viper-to-Boogie translation, and discusses other aspects of the existing Viper-to-Boogie translation that are relevant for the rest of this chapter. Section 3.4 introduces our general forward simulation methodology for relating front-end statements with IVL statements; the section presents the general concepts on Viper and Boogie statements. Section 3.5 presents how we formally validate the existing Viper-to-Boogie implementation using our forward simulation methodology. Section 3.6 evaluates the certificates generated by our instrumentation. Section 3.7 presents related work and Section 3.8 discusses the impact this chapter had on the Viper ecosystem. Finally, Section 3.9 discusses directions for future work.

Access to tool and Isabelle formalisation

Our certificate-producing version of the Viper-to-Boogie implementation is available online:

- Repository:
  `https://github.com/viperproject/carbon-proofgen`
- Branch for dissertation: dissertation-gaurav
- Commit hash at time of dissertation submission:
  3b4215d2db55a2af46265cd801e678fc9410e50f

The main code for generating certificates is in the *src/main/scala/viper/-carbon/proofgen* folder, which contains exclusively new code added by us. The existing Viper-to-Boogie implementation itself is spread across the remaining subfolders of *src/main/scala/viper/carbon*. For instance, *src/main/scala/viper/carbon/Carbon.scala* contains the entry point of the implementation and *src/main/scala/viper/carbon/modules* contains the code implementing different parts of the translation. We have instrumented parts of these folders containing the existing Viper-to-Boogie implementation in order to obtain sufficient information to generate certificates. The number of lines of added code for the instrumentation (fewer than 500 lines) is significantly smaller than the code that actually generates certificates (the latter is in *src/main/scala/viper/carbon/proofgen*).

[103]: Parthasarathy (2024), *Viper Semantics and Certificate Metatheory Formalisation*

Our formal Viper semantics and the metatheory used by the certificates (both expressed in Isabelle) is available online [103]:

- Repository:
  `https://github.com/viperproject/viper-roots`
- Branch for dissertation: dissertation-gaurav
- Commit hash at time of dissertation submission:
  845f8eed90c6dd51fad7794a98bab4a5bf5a598d

The folder *vipersemcommon* contains the syntax of the Viper language as well as the formalisation of some basic concepts, and the folder *viper-total-heaps* contains the main parts of the formalisation relevant for this chapter.

[104]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language – Artifact*

[60]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language*

Finally, our publicly-available artifact [104] that supplements our corresponding conference publication associated with this chapter [60] is also available online:

`https://zenodo.org/records/10802176`

This artifact contains older snapshots of our certificate-producing Viper-to-Boogie implementation and our Isabelle formalisation (but still relatively up-to-date versions). Moreover, this artifact includes instructions for exploring our certificate-producing implementation and our Isabelle formalisation, and includes instructions for reproducing our evaluation.

3.2. A Formal Semantics for Viper

In this section, we first present the Viper subset relevant for this dissertation (Subsection 3.2.1) and show a Viper example (Subsection 3.2.2). Next, we present a novel operational semantics for this Viper subset (Subsection 3.2.3, Subsection 3.2.4, Subsection 3.2.5), which we have mechanised in Isabelle, and then present correctness of a Viper program in terms of this semantics (Subsection 3.2.6). Then, we illustrate the semantics on a Viper example (Subsection 3.2.7). Finally, we discuss design

$$\begin{aligned}
\mathit{VUnaryOp} \ni \mathit{uop} &::= \ - \mid \,! \\
\mathit{VBinaryOp} \ni \mathit{bop} &::= \ == \mid \,!= \mid + \mid - \mid * \mid /_{\mathsf{Int}} \mid /_{\mathsf{Perm}} \mid \mathsf{mod} \mid \leq \mid < \mid \geq \mid > \mid \&\& \mid \,||\, \mid \Rightarrow \\
\mathit{VExpr} \ni e &::= x \mid \mathtt{true} \mid \mathtt{false} \mid i \mid \mathtt{write} \mid \mathtt{none} \mid \mathtt{null} \mid e.f \mid e \ \mathit{bop}\ e \mid \mathit{uop}(e) \mid \\
&\quad\ e\ ?\ e : e \mid \mathtt{perm}(e.f) \\
\mathit{VAssert} \ni A &::= e \mid \mathtt{acc}(e.f, e) \mid A \ \&\&\ A \mid e \Rightarrow A \mid e\ ?\ A : A \\
\mathit{VType} \ni \tau &::= \mathsf{Int} \mid \mathsf{Bool} \mid \mathsf{Ref} \mid \mathsf{Perm} \\
\mathit{VStmt} \ni s &::= x := e \mid e.f := e \mid \vec{y} := m(\vec{es}) \mid m(\vec{es}) \mid \mathtt{inhale}\ A \mid \mathtt{exhale}\ A \mid \mathtt{assert}\ A \mid \\
&\quad\ \mathtt{var}\ x : \tau\ ;\ s \mid s; s \mid \mathtt{if}\ (e)\ \{s\}\ \mathtt{else}\ \{s\} \\
\mathit{VMethodDecl} \ni \mathit{methodDecl} &::= \ \mathtt{method}\ m(\overrightarrow{x : \tau})\ \mathtt{returns}\ (\overrightarrow{y : \tau}) \\
&\quad\ \ \mathtt{requires}\ A \\
&\quad\ \ \mathtt{ensures}\ A \\
&\quad\ \ \{\ s\ \} \\
\mathit{VFieldDecl} \ni \mathit{fieldDecl} &::= \mathtt{field}\ f : \tau \\
\mathit{VProg} \ni \mathit{prog} &::= \overrightarrow{\mathit{fieldDecl}}; \overrightarrow{\mathit{methodDecl}}
\end{aligned}$$

Figure 3.1: The syntax of our formalised Viper subset. m (method name) and f (field name) denote Viper identifiers. x and y denote variables. i denotes an integer constant. `write` and `none` denote the full permission amount (*i.e.* 1) and the empty permission amount (*i.e.* 0), respectively. Note that Viper differentiates between two divisions: integer division and permission division. The former is the Euclidean division between two integers. The latter is the standard division between two reals (permission amounts in Viper are represented by reals). In practice, the notation / is used for both and Viper interprets the division based on the type information, defaulting to real division in ambiguous cases; the notation \ (not used here) forces integer division. In our formalisation, we use separate syntax for the two divisions. Finally, one can encode Viper's support for multiple preconditions (resp. postconditions) in our formalisation by taking the conjunction of the preconditions (resp. postconditions).

decisions taken for our semantics in Subsection 3.2.8. The presented Viper semantics forms the basis for our certificates that formally validate the existing Viper-to-Boogie implementation. We discuss the relationship of our operational Viper semantics to other Viper formalisations as part of our discussion on related work (Section 3.7).

3.2.1. The Viper Language

Our supported syntax for Viper programs is shown in Figure 3.1. Viper is an imperative language, which in the subset considered here consists of a list of top-level declarations of fields and methods. Reference-field pairs are used to access Viper's built-in heap. Field declarations specify the type of the values stored for corresponding reference-field pairs in the heap. In our subset, we support integers, Booleans, references, and permission amounts (whose meaning will become clear below). Each Viper method has a specification given by a pre- and postcondition, which are Viper assertions. The body of each Viper method is a Viper statement and calls to a method m within a method body are treated modularly w.r.t. m's specification instead of executing the callee's body. In addition to statements, Viper separates assertions from expressions.

Viper's verification methodology employs a custom advanced program logic, in this case based on a flavour of separation logic (SL) called *implicit dynamic frames* (IDF) [33, 101], which reasons about the heap via *permissions*. Viper uses a *fractional permission* model [105] of IDF, which

[33]: Smans et al. (2012), *Implicit Dynamic Frames*
[101]: Parkinson et al. (2012), *The Relationship Between Separation Logic and Implicit Dynamic Frames*

[105]: Boyland (2003), *Checking Interference with Fractional Permissions*

associates fractional permission amounts that range between 0 and 1 with heap locations; nonzero permission is required to *read* heap locations and full (*i.e.* 1) permission is required to *write to* heap locations. Viper states explicitly track the currently held permission amount for each heap location.

In the following, we first discuss Viper assertions. These are essentially IDF assertions, which include the specification of permissions. Then, we discuss Viper expressions, and Viper statements.

Assertions and expressions

The *accessibility predicate* **acc**$(e.f, p)$ represents a *resource* (a logical notion which can be neither freely fabricated nor duplicated): the fractional (p) amount of *permission to access heap location* $e.f$.[1] The assertion A && B expresses the IDF generalisation of the *separating conjunction* from SL, which specifies that the permissions in A and B must *sum up to an amount currently held*. In particular, as in SL, since the amount to each heap location can be at most 1, this means that `acc(x.f,p) && acc(y.f,q)` is satisfied only in states in which either `p` + `q` is at most 1 or in which `x` and `y` are different. Other assertions include Boolean expressions, implications where the left-hand side is a Boolean expression, and conditional assertions $e \ ? \ A : B$, which denotes A if e evaluates to true in a given Viper state and otherwise denotes B.

1: For readers familiar with separation logics, this is analogous to a fractional *points-to assertion* in a separation logic.

One difference between IDF and SL is that IDF (and thus, Viper) supports *heap-dependent* expressions. In particular, in Viper, heap locations can be read via a field access $e.f$, where e is an expression that evaluates to a reference and f is a field. This means Viper supports expressions such as `x.val == 5` or `x.f.f`, whose evaluation is *partial* (only allowed with suitable permissions). This necessitates a notion of *well-definedness* checks on expressions to ensure that those expressions are only evaluated when suitable permissions are held; we will discuss these in Subsection 3.2.4.

Field accesses express values of heap locations, while accessibility predicate specify permissions to heap locations. This distinction is different from SL, where *points-to assertions* specify the permission to a heap location *and* its value. As a result, IDF supports separating conjunctions A && B where B expresses constraints on a heap location via the corresponding field access for which A already specifies full permission. For instance, `acc(x.f, write) && x.f >= 0` is a typical Viper assertion (**write** denotes the full permission amount, that is, permission amount 1). In SL, one would express the analogous assertion as $\exists v.$ `x.f` $\mapsto v * v \geq 0$;[2] the surrounding existential quantifier must be used to express `x.f`'s value in both the points-to assertion and the second conjunct.

2: `x.f` $\mapsto v$ is the SL points-to assertion denoting full permission to `x.f` and expressing that `x.f` stores value v. The symbol $*$ is the SL separating conjunction.

Apart from field accesses and other basic constructs, Viper expressions include *permission introspection* **perm**$(e.f)$, which evaluates to the permission *currently held* at the heap location $e.f$; its precise semantics will become clear in Subsection 3.2.4 and Subsection 3.2.5. Permission introspection is used by Viper front-ends to encode proof search algorithms and to encode proof obligations. An example of the former is to use permission introspection to branch on the availability of permissions (*e.g.* **if** (**perm**$(e.f)$ == **write**) $\{s_1\}$ **else** $\{s_2\}$), where the statement s_1 may encode the application of a proof rule that requires full permission to $e.f$.

An example of the latter is **assert** perm($e.f$) == **none** (**none** denotes the empty permission amount, that is, permission amount 0), which checks that there is no permission left to $e.f$ and fails otherwise (required if a proof rule demands that no permission to $e.f$ is leaked).[3]

3: Front-ends usually model leak checks using Viper's **forperm** construct, which is a variant of permission introspection that quantifies over a set of heap locations for which there is nonzero permission. Our subset does not cover this variant; it could be simply encoded if we extended our subset to include quantifiers.

Statements

Basic Viper primitive statements include variable assignments, field assignments, and method calls. For method calls, there is a distinction depending on whether the method returns results in which case the results are stored in target variables. In terms of control flow, Viper supports scoped variable declarations (*i.e.* **var** $x : \tau\ ;\ s$), sequential composition and conditional branching. For a scoped variable declaration **var** $x : \tau\ ;\ s$, the variable x is nondeterministically assigned a value of type τ before executing s.

Viper uses two main statement primitives to encode separation logic reasoning: (1) **inhale** A adds the permissions specified by assertion A to the state and assumes the logical constraints in A, (2) **exhale** A *removes* the permissions specified by A, and *fails* if a constraint in A does not hold.[4] **inhale** and **exhale** operations are typically used in Viper to encode external or more-complex operations. Moreover, as we will see, method calls are expressed in the semantics by exhaling the precondition and then inhaling the postcondition of the callee. If the method returns results, one must additionally assign values nondeterministically for the target variables before inhaling the postcondition.

4: For separation-logic-versed readers, the Hoare triples $\{R\}$ **inhale** A $\{R * A\}$ and $\{R * A\}$ **exhale** A $\{R\}$ reflect this behaviour (assuming the expressions in A and R are well-defined).

Finally, **assert** A checks if **exhale** A would succeed in the current state. If it does, it leaves the state unchanged, and otherwise fails. This primitive is encodable via the remaining subset:[5]

```
var b: Bool;
if(b) { exhale A; inhale false } else { exhale true }
```

5: This means replacing **assert** A with the encoding does not change the correctness of the program.

Nevertheless, we include it as a separate construct, since (1) the source language supports it, and (2) Viper tools (including the Viper-to-Boogie implementation) treat **assert** differently than the encoded version.

The significance of permission introspection

Our certificate-producing version of the Viper-to-Boogie implementation generates certificates for the full subset shown in Figure 3.1 *except for* permission introspection. We nevertheless formalised permission introspection for two reasons. First, the presence of permission introspection has a significant impact on the Viper formalisation of other features included in our subset (and features not supported in our subset). In particular, permission introspection has an impact on how to formalise the evaluation of expressions and the semantics of **inhale** and **exhale**, as we will discuss later in this section. Moreover, permission introspection also impacts how to formalise the correctness of a Viper program. Second, one of our goals is to ensure the extensibility of our certification approach to larger Viper subsets (as mentioned at the end of Section 3.1). These two points together were the reason why we included permission introspection in our semantics. This way adding certificate-producing support for permission introspection need not change the semantics of other features in any way. In fact, given our current semantics, adding certificate-producing support for permission introspection would be straightforward and similar to our certification of field accesses.

The decision to include permission introspection also led us to explore its behaviour in combination with other Viper features. As a result, we observed two new insights: (1) the existing Viper-to-Boogie implementation was originally not correctly handling permission introspection in combination with `exhale`, which we then fixed, and (2) the semantics of *unfolding expressions* (a feature outside of our subset) combined with permission introspection is unclear. These insights further show the impact permission introspection has on the semantics of Viper. We discuss these insights in more detail in Section 3.8.

Unsupported features

The main Viper features *not* included in our formalised subset (and thus also not supported by our certificate-producing Viper version) are: loops, (labelled) old expressions, quantifiers, more-complex resource assertions (predicates, magic wands, iterated separating conjunctions), heap-dependent functions, and domains. For most of these features, once the semantics is formalised, the generation of certificates should be similar to the generation of certificates for features that we already support and thus the general methodology developed in this dissertation should be directly applicable. We discuss the extension of our work to unsupported features as part of future work (Section 3.9).

Comparison to Boogie

A fundamental difference between Viper and Boogie is that Viper has a built-in mutable heap and Boogie does not. Reasoning about a mutable heap is complex, for instance, because of aliasing: updating a heap location `x.f` affects the value stored at heap location `y.f` if `x` and `y` evaluate to the same value (*i.e.* if they are aliases). Such aliasing issues do not arise if one just tracks variables and no heap such as Boogie. As a result, Viper supports heap reasoning via implicit dynamic frames, while Boogie uses a simpler logic whose assertions (*i.e.* expression evaluating to Booleans) are essentially first-order logic formulas. One consequence is the large difference in the two state models. A Viper state consists of a variable store, a heap (mapping heap locations to current values) and a *permission mask* (mapping heap locations to current permission amounts), while a Boogie state simply tracks variable stores. Another consequence is that Viper's evaluation of expressions is *partial* (*e.g.* heap accesses must be checked to be well-defined), while Boogie's evaluation of (well-typed) expressions is total.

The Viper statement representation used by the Viper verifier implementation and the Boogie statement representation used by the Boogie verifier implementation are structured differently (see Figure 2.27 on page 80 in Chapter 2 for the Boogie statement representation). These are the representations, which we have formalised, and must formally connect. The Viper statement representation uses a standard sequential

```
field f: Int
field g: Int

method main()
{
    var x: Ref
    inhale acc(x.f)
    var y: Ref
    inhale acc(y.g)
    x.f := 0
    y.g := 100
    sum(x, y, 0)
    assert x.f == 5050
}

method sum(x: Ref, y: Ref, i: Int)
    requires acc(x.f) && acc(y.g, 1/2) &&
             x.f == (i * (i - 1)) / 2 && i <= y.g + 1
    ensures  acc(x.f) && acc(y.g, 1/2) &&
             x.f == (y.g * (y.g + 1)) / 2
{
    if(i <= y.g) {
        x.f := x.f + i
        sum(x, y, i+1)
    }
}
```

Figure 3.2: An example of a correct Viper program. `main` has the trivial pre- and postcondition `true`, since `main` does not explicitly specify a pre- and postcondition. `acc(x.f)` is syntactic sugar for `acc(x.f, write)` (full permission). All divisions in this example are integer divisions.

composition $s_1; s_2$ to compose two statements s_1 and s_2, whereas the Boogie statement representation is given by a list of *statement blocks*. Each statement block consists of a list of basic commands (*i.e.* no control flow), followed by an optional control-flow element. Our validation generates certificates that directly relates these representations and thus must bridge this gap.

Other differences include that Viper supports scoped variables, while in Boogie all local variables must be declared at the beginning of a procedure. Moreover, Boogie supports quantification over *types*, which Viper does not support. Finally, in our particular subset, Viper programs cannot quantify over *values* and cannot express background axiomatisations, both of which Boogie supports. However, both of these are supported by Viper features outside of our subset.

3.2.2. Viper Example

To make the Viper language more concrete, consider the Viper program shown in Figure 3.2. This program has two field declarations and two Viper methods. The method `sum` adds the sum of all integers between `i` and `y.g` to `x.f` and is implemented recursively to deal with this statically unbounded range. The method `main` first inhales full permission to `x.f` and `y.g`. Then, `main` stores appropriate values in `x.f` and `y.g`, and then calls `sum` with the goal of storing the sum of the first 100 integers into `x.f`. The two **`inhale`** statements justify writing to `x.f` and `y.g`. The final **`assert`** statement checks whether `x.f` indeed stores this sum (which is given by 5050).

This Viper program in Figure 3.2 is *correct*, which intuitively means that there is no *failing* Viper method execution (for instance, the **`assert`** statement in `main` does not fail). We will make the notion of correctness precise after introducing our operational semantics for Viper programs, and will then explain why this program is indeed correct in Subsection 3.2.7.

3.2.3. Operational Semantics: Values and State Model

We formalise Viper values using the following Isabelle algebraic data type vprval:

$$\begin{aligned}\mathsf{vprval} &\triangleq \mathsf{VIntVal}(\mathsf{int}_{isa}) \mid \mathsf{VBoolVal}(\mathsf{bool}_{isa}) \mid \\ &\quad\; \mathsf{VRefVal}(\mathsf{ref}) \mid \mathsf{VPermVal}(\mathsf{real}_{isa}) \\ \mathsf{ref} &\triangleq \mathsf{Null} \mid \mathsf{Address}(\mathsf{nat}_{isa})\end{aligned}$$

There are four kinds of values: integers, Booleans, references, and permission amounts. Integer and Boolean values are embedded via their Isabelle counterparts (*e.g.* int_{isa} denotes the Isabelle integer type). Permission amounts are embedded via the Isabelle type for reals.[6] Finally, references are are modelled via another Isabelle algebraic data type ref shown above. References are either null or an address whose identifier is given by a natural number.[7]

6: One reason why Viper uses reals for permission amounts instead of the more traditional rationals is that SMT solvers have built-in support for reals but usually not for rationals.

7: Addresses should be represented by a type with infinitely many values, since there is no single finite bound feasible for every Viper program. There is no fundamental reason for choosing natural numbers; our generated certificates do not rely on the underlying type used for the addresses.

Viper programs can be understood in terms of the sets of possible executions through each Viper method body. Analogously to our formalisation of Boogie in Chapter 2, we distinguish three possible outcomes for finite Viper executions: *failure* if the execution fails, *magic* if the execution stops (*i.e. goes to magic*), and a *normal outcome* if the execution succeeds (*i.e.* terminates normally) and transitions to a state because neither of the other two cases occur. The three outcomes are represented formally by the following algebraic data type:

$$\mathsf{outcome}_v \triangleq \mathsf{F} \mid \mathsf{M} \mid \mathsf{N}(\mathsf{state}_v)$$

where (1) F denotes a *failure outcome*, (2) M denotes a *magic outcome*, and (3) $\mathsf{N}(\sigma_v)$ denotes a *normal outcome*, where σ_v is the resulting Viper state (state_v is the corresponding type representing such states). A Viper state σ_v is a triple (st, h, m), where *st* is a store (a partial mapping from variables to values), h is a heap (a total mapping from heap locations to values), and m is a permission mask (a total mapping from heap locations to nonnegative permission amounts). We use projection functions to map a state σ_v to its three components: $\mathsf{ST}(\sigma_v)$ for the store, $\mathsf{H}(\sigma_v)$ for the heap, $\Pi(\sigma_v)$ for the permission mask.

Implicit dynamic frames can be expressed with heaps represented by total mappings or partial mappings. The motivation for choosing a total mapping for our semantics is because the Viper-to-Boogie implementation encodes the heap in Boogie via a total mapping (moreover, a total mapping is a common choice for implicit dynamic frames formalisations). We elaborate on this choice in Subsection 3.2.8.

Determining the outcome of a Viper execution is more involved than for a Boogie execution. In Boogie, the failure and magic outcomes are reached only if a Boolean expression of an `assume` or `assert` command evaluates to false. In Viper, these outcomes are reached in many more cases, as we will see. For instance, a failure outcome is reached when a Viper expression is ill-defined or there is insufficient permission to write to a heap location, and a magic outcome is reached whenever an `inhale` would lead to an *inconsistent* state. A Viper state is *consistent* if its permission mask is consistent, *i.e.* maps each location to values between 0 and 1. This ensures, for instance, that inhaling `acc(x.f, write) && acc(y.f, write)` goes to magic if `x` and `y` evaluate to the same reference, which correctly captures the separating conjunction.

3.2.4. Operational Semantics: Expression Evaluation

Formalising expression evaluation requires care in Viper for two reasons. First, in a given state, not even all type-correct expressions are *well-defined*: in our subset this can be either because of (1) division by zero, or (2) dereferencing a heap location for which no permission is held (subsuming null dereferences). In our semantics, evaluating an ill-defined expression causes execution to fail (in contrast to Boogie, where expression evaluation cannot fail).[8] Second, when evaluating expressions as part of an `exhale` operation, one must check whether sufficient permission is held in a *potentially different state* than the state in which the expression is evaluated. As a result, our expression evaluation judgement depends on *two* states. As we will make clear in Subsection 3.2.5, our semantics requires this differentiation because of the potential combination of `exhale` and permission introspection. The two states are always the same whenever expressions are evaluated which are not part of an `exhale` operation.

8: Boogie also has division, but defines division by 0 to be some fixed but unspecified value.

Our expression evaluation judgement $\sigma_v^0 \vdash \langle e, \sigma_v \rangle \Downarrow_{\mathsf{V}} r_v$ expresses that expression e evaluates to the result r_v in state σ_v where the permission checks are performed in state σ_v^0. We call σ_v the *evaluation state* and σ_v^0 the *permission definedness state*, since the latter is used only to check whether there is sufficient permission. r_v is either a normal result $\mathsf{V}(v)$ where v is a Viper value or it is a failure result $\lightning$. In the former case, the expression is well-defined (and evaluates to value v) and in the latter case it is not. We lift the judgement for a single expression to a list of expressions es, which yields the judgement $\sigma_v^0 \vdash \langle es, \sigma_v \rangle [\Downarrow]_{\mathsf{V}} r_v$, where r_v is $\mathsf{V}(vs)$ and vs is the list of values to which es evaluate to or r_v is $\lightning$ if an expression in es is ill-defined.

The expression evaluation judgements for a single expression and a list of expressions are defined in a mutually-inductive way. A selection of rules of the inductive definition is shown in Figure 3.3. The full formalisation is available online [103]. As the rules FIELD and FIELD-NULL show, a field access fails if there is no permission to the corresponding heap location in the permission definedness state or the receiver evaluates to null. If there is permission, then the heap value is looked up in the evaluation state.

[103]: Parthasarathy (2024), *Viper Semantics and Certificate Metatheory Formalisation*

$$\frac{\sigma_v^0 \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(\mathsf{Address}(a)) \qquad r_v = \begin{array}{l} \mathit{if}\ \Pi(\sigma_v^0)(a,f) > 0 \\ \mathit{then}\ \mathsf{V}(\mathsf{H}(\sigma_v)(a,f)) \\ \mathit{else}\ \lightning \end{array}}{\sigma_v^0 \vdash \langle e.f, \sigma_v\rangle \Downarrow_{\mathsf{v}} r_v}\ (\textsc{field})$$

$$\frac{\sigma_v^0 \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(\mathsf{VRefVal}(\mathsf{Null}))}{\sigma_v^0 \vdash \langle e.f, \sigma_v\rangle \Downarrow_{\mathsf{v}} \lightning}\ (\textsc{field-null})$$

$$\frac{\sigma_v^0 \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(\mathsf{VRefVal}(\mathsf{Address}(a)))}{\sigma_v^0 \vdash \langle \mathtt{perm}(e.f), \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(\mathsf{VPermVal}(\Pi(\sigma_v)(a,f)))}\ (\textsc{perm})$$

$$\frac{\sigma_v^0 \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(\mathsf{VRefVal}(\mathsf{Null}))}{\sigma_v^0 \vdash \langle \mathtt{perm}(e.f), \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(\mathsf{VPermVal}(0))}\ (\textsc{perm-null})$$

$$\frac{\begin{array}{c}\sigma_v^0 \vdash \langle e_1, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(v_1) \\ \mathsf{binopLazyEval}(v_1, \mathit{bop}) = \mathsf{Some}(v')\end{array}}{\sigma_v^0 \vdash \langle e_1\ \mathit{bop}\ e_2, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(v')}\ (\textsc{bop-lazy})$$

$$\frac{\begin{array}{c}\sigma_v^0 \vdash \langle e_1, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(v_1) \\ \mathsf{binopLazyEval}(v_1, \mathit{bop}) = \mathsf{None} \\ \sigma_v^0 \vdash \langle e_2, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(v_2) \\ v_1\ \overline{\mathit{bop}}\ v_2 = \mathsf{Some}(r_v)\end{array}}{\sigma_v^0 \vdash \langle e_1\ \mathit{bop}\ e_2, \sigma_v\rangle \Downarrow_{\mathsf{v}} r_v}\ (\textsc{bop-eager})$$

$$\frac{\begin{array}{c}\sigma_v^0 \vdash \langle e_1, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(v_1) \\ \mathsf{binopLazyEval}(v_1, \mathit{bop}) = \mathsf{None} \\ \sigma_v^0 \vdash \langle e_2, \sigma_v\rangle \Downarrow_{\mathsf{v}} \lightning \\ \exists v_2, r_v.\ v_1\ \overline{\mathit{bop}}\ v_2 = \mathsf{Some}(r_v)\end{array}}{\sigma_v^0 \vdash \langle e_1\ \mathit{bop}\ e_2, \sigma_v\rangle \Downarrow_{\mathsf{v}} \lightning}\ (\textsc{bop-eager-fail})$$

$$\frac{\begin{array}{c}\mathit{es} = \mathsf{defineSubExprsE}(e) \quad \mathit{es} \neq [] \\ \sigma_v^0 \vdash \langle \mathit{es}, \sigma_v\rangle\ [\Downarrow]_{\mathsf{v}}\ \lightning\end{array}}{\sigma_v^0 \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \lightning}\ (\textsc{exp-subexp-fail})$$

$$\frac{}{\sigma_v^0 \vdash \langle [], \sigma_v\rangle\ [\Downarrow]_{\mathsf{v}}\ \mathsf{V}([])}\ (\textsc{exps-nil})$$

$$\frac{\sigma_v^0 \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \lightning}{\sigma_v^0 \vdash \langle e :: \mathit{es}, \sigma_v\rangle\ [\Downarrow]_{\mathsf{v}}\ \lightning}\ (\textsc{exps-f})$$

$$\frac{\begin{array}{c}\sigma_v^0 \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(v) \\ \sigma_v^0 \vdash \langle \mathit{es}, \sigma_v\rangle\ [\Downarrow]_{\mathsf{v}}\ r_v \\ (r_v = \lightning \wedge r_v' = \lightning) \vee (\exists \mathit{vs}'.\ r_v = \mathsf{V}(\mathit{vs}') \wedge r_v' = \mathsf{V}(v :: \mathit{vs}'))\end{array}}{\sigma_v^0 \vdash \langle e :: \mathit{es}, \sigma_v\rangle\ [\Downarrow]_{\mathsf{v}}\ r_v'}\ (\textsc{exps-cons})$$

Figure 3.3: Selected rules for the definition of expression evaluation judgements for a single expression and an expression list. The term $[]$ denotes the empty list and the term $e :: \mathit{es}$ denotes the list obtained by prepending e to the list es. The term $\overline{\mathit{bop}}$ denotes the semantic interpretation of a binary operation bop if the operation is well-typed and well-defined for the given arguments (*i.e.* in this case, returning $\mathsf{Some}(\mathsf{V}(v))$ where v is the resulting value). Moreover, $\overline{\mathit{bop}}$ returns $\mathsf{Some}(\lightning)$ if the operation is well-typed but ill-defined (*i.e.* division or modulo by 0) and returns None if the operation is not well-typed for the given arguments. defineSubExprsStmt and binopLazyEval are defined in Figure 3.4.

The rule EXP-SUBEXP-FAIL catches the case when a *definedness subexpression* of expression e, given by $\mathsf{defineSubExprsE}(e)$ in Figure 3.3, is ill-defined, which results in e being ill-defined. The main purpose of defining the notion of a definedness subexpression is to express the EXP-SUBEXP-FAIL

$$
\begin{array}{ll}
\mathsf{defineSubExprsE}(e) \triangleq & \mathsf{case}\ e\ \mathsf{of} \\
& \quad uop(e) \Rightarrow [e] \\
& \quad \mid e_1\ bop\ e_2 \Rightarrow [e_1] \\
& \quad \mid e.f \Rightarrow [e] \\
& \quad \mid e_1\ ?\ e_2 : e_3 \Rightarrow [e_1] \\
& \quad \mid \mathtt{perm}(e.f) \Rightarrow [e] \\
& \quad \mid \_ \Rightarrow [] \\
\mathsf{binopLazyEval}(v_1, bop) \triangleq & \mathsf{case}\ (v_1, bop)\ \mathsf{of} \\
& \quad \mid (\mathsf{VBoolVal}(\mathsf{true}), \mid\mid) \Rightarrow \mathsf{Some}(\mathsf{VBoolVal}(\mathsf{true})) \\
& \quad \mid (\mathsf{VBoolVal}(\mathsf{false}), \&\&) \Rightarrow \mathsf{Some}(\mathsf{VBoolVal}(\mathsf{false})) \\
& \quad \mid (\mathsf{VBoolVal}(\mathsf{false}), \Rightarrow) \Rightarrow \mathsf{Some}(\mathsf{VBoolVal}(\mathsf{true})) \\
& \quad \mid \_ \Rightarrow \mathsf{None}
\end{array}
$$

Figure 3.4: Auxiliary definitions for the expression evaluation judgement. Note that $\mathsf{defineSubExprsE}(e)$ returns a list of expressions, even though in our subset the returned list has at most one element. This choice is motivated by our goal of making our work extensible to larger Viper subsets. For expressions outside our subset, the returned list would have more elements. Thus, our definition here can be expanded naturally to those.

rule, which catches *in a single rule* the cases when an expression e fails due to failure of a subexpression of e, instead of having to define a separate rule for different kinds of expressions. For example, the receiver of e is a definedness subexpression of the field access $e.f$, because if e fails to evaluate, then so does $e.f$. Note that e_2 is *not* considered a definedness subexpression of $e_1\ bop\ e_2$. The reason is that the evaluation of certain binary operations in Viper is lazy, and as a result, the binary operation is well-defined in certain cases even if e_2 is ill-defined. For instance, $e_1 \Rightarrow e_2$ evaluates to true if e_1 evaluates to false even if e_2 is not well-defined. As a result, for binary operations there is a separate rule bop-eager-fail for dealing with the case when e_2 is ill-defined. bop-eager-fail makes sure that the binary evaluation is eager (*i.e.* $\mathsf{binopLazyEval}(v_1, bop) = \mathsf{None}$). Note that in bop-eager-fail, the reduction is defined only if the first operand evaluates to a value v_1 whose type is compatible with the binary operation bop (*i.e.* $\exists v_2, r_v.\ v_1\ \overline{bop}\ v_2 = \mathsf{Some}(r_v)$). For instance, if v_1 were an integer value, then the rule would not apply for Boolean binary operations. This choice does not affect well-typed programs. The motivation for this choice is to simplify the generated certificates, since this choice reduces the amount of well-typedness proofs required in a certificate for concrete expressions in the corresponding Boogie program.

The rules bop-lazy and bop-eager capture the evaluation of binary operations when the two operands are well-defined. bop-lazy captures the case when a binary operation is evaluated lazily (*i.e.* the evaluation is determined by the first operand, thus *short-circuiting* the evaluation). A binary operation bop is evaluated lazily to value v' iff the first operand evaluates to v_1 and $\mathsf{binopLazyEval}(v_1, bop) = \mathsf{Some}(v')$. bop-eager captures the case when a binary operation is evaluated eagerly. Note that bop-eager ensures that division by 0 results in failure (*e.g.* $\mathsf{VIntVal}(i)\ \overline{/}_{\mathsf{Int}}\ \mathsf{VIntVal}(0) = \mathsf{Some}(\lightning)$ for all integers i).

Finally, consider the rules perm and perm-null that model the evaluation of permission introspection $\mathtt{perm}(e.f)$. $\mathtt{perm}(e.f)$ evaluates to the permission of the corresponding heap location in the *evaluation state*. Contrary to field accesses, permission introspection is not ill-defined if the receiver evaluates to null. In particular, if the receiver evaluates to null, then permission introspection evaluates to 0. Note that this semantics of `perm(null.f)` may be somewhat unintuitive compared to the semantics of field accesses, which requires the receiver to be non-null. We chose to evaluate `perm(null.f)` to 0, because this is how the Viper verifier implementation treats this case. Adjusting the semantics and the verifier implementation to require the receiver to be non-null would also be an option.

3.2.5. Operational Semantics: Statement Reduction

We give a big-step operational semantics to Viper statements. The judgement $\Gamma_v \vdash \langle s, \sigma_v \rangle \rightarrow_{\mathsf{V}} r_v$ holds if in the *Viper context* Γ_v the execution of statement s in the state σ_v terminates with outcome r_v. A Viper context Γ_v contains three components: (1) $\mathsf{Vars}(\Gamma_v)$: a partial mapping from the in-scope variable names to their declared types, (2) $\mathsf{Fields}(\Gamma_v)$: a partial mapping from field names in the program to their declared types, (3) $\mathsf{Methods}(\Gamma_v)$: a partial mapping from method names in the program to their declared method signatures. The in-scope variable mapping changes in the execution reduction within the body of a scoped variable statement, while the other two mappings always remain the same during the execution reduction.

The judgement is defined inductively via the rules shown in Figure 3.5. As in the Boogie semantics, the rules ensure that well-typed states remain well-typed. For instance, the rules FIELD-SUCC and ASSIGN for the field and variable assignments do not reduce if the to-be-assigned value reduces to a value that does not match the corresponding declared type of the field or variable. This choice has no effect on well-typed programs.[9] Rule FIELD-SUCC additionally reflects that full permission is required to write to a heap location, otherwise the field assignment fails (the failure case is captured by rule FIELD-FAIL). Analogously to the expression evaluation formalisation, there is a single rule that captures the failure case when a definedness subexpression of a statement is ill-defined (rule STMT-SUBEXP-FAIL).

9: To gain confidence that these extra conditions do not rule out desired executions, one could prove a separate type soundness result. We have not proved such a result, but doing so would be straightforward.

Inhale and exhale

The most interesting statements are the `inhale` and `exhale` primitives. Recall that (1) `inhale` A adds the permissions specified by assertion A to the state and assumes the logical constraints in A, and (2) `exhale` A removes the permissions specified by A, and fails if a constraint in A does not hold. We define their semantics via separate judgements, which we will discuss below in more detail. First, we will discuss two high-level decisions for the semantics: (1) our semantics of `inhale` A leaves the heap unchanged, while `exhale` A causes the loss of heap information for those heap locations for which the `exhale` removes all available permission, and (2) our semantics of `inhale` A and `exhale` A are both expressed operationally by decomposing the assertion A from left to right. Let us discuss both of these decisions.

$$\frac{\begin{array}{c}\sigma_v \vdash \langle e_r, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(\mathsf{VRefVal}(\mathsf{Address}(a))) \\ \sigma_v \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(v) \\ \Pi(\sigma_v)(a,f) = 1 \\ \mathsf{Fields}(\Gamma_v)(f) = \mathsf{typVpr}(v) \\ \sigma_v' = (\mathsf{ST}(\sigma_v), \mathsf{H}(\sigma_v)((a,f) \mapsto v), \Pi(\sigma_v))\end{array}}{\Gamma_v \vdash \langle e_r.f := e, \sigma_v\rangle \longrightarrow_{\mathsf{v}} \mathsf{N}(\sigma_v')}\ (\textsc{field-succ})$$

$$\frac{\begin{array}{c}\sigma_v \vdash \langle e_r, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(\mathsf{VRefVal}(r)) \\ \sigma_v \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(v) \\ r = \mathsf{Null} \vee (\exists a.\ r = \mathsf{Address}(a) \wedge \Pi(\sigma_v)(a,f) < 1)\end{array}}{\Gamma_v \vdash \langle e_r.f := e, \sigma_v\rangle \longrightarrow_{\mathsf{v}} \mathsf{F}}\ (\textsc{field-fail})$$

$$\frac{\begin{array}{c}\sigma_v \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(v) \\ \mathsf{Vars}(\Gamma_v)(x) = \mathsf{typVpr}(v) \\ \sigma_v' = (\mathsf{ST}(\sigma_v)(x \mapsto v), \mathsf{H}(\sigma_v), \Pi(\sigma_v))\end{array}}{\Gamma_v \vdash \langle x := e, \sigma_v\rangle \longrightarrow_{\mathsf{v}} \mathsf{N}(\sigma_v')}\ (\textsc{assign}) \qquad \frac{\langle A, \sigma_v\rangle \longrightarrow_{\mathsf{inh}} r_v}{\Gamma_v \vdash \langle \mathtt{inhale}\ A, \sigma_v\rangle \longrightarrow_{\mathsf{v}} r_v}\ (\textsc{inh})$$

$$\frac{\begin{array}{c}\sigma_v \vdash \langle A, \sigma_v\rangle \longrightarrow_{\mathsf{rc}} \mathsf{N}(\sigma_v') \\ \mathsf{nonDet}(\sigma_v, \sigma_v', \sigma_v'')\end{array}}{\Gamma_v \vdash \langle \mathtt{exhale}\ A, \sigma_v\rangle \longrightarrow_{\mathsf{v}} \mathsf{N}(\sigma_v'')}\ (\textsc{exh-succ}) \qquad \frac{\sigma_v \vdash \langle A, \sigma_v\rangle \longrightarrow_{\mathsf{rc}} \mathsf{F}}{\Gamma_v \vdash \langle \mathtt{exhale}\ A, \sigma_v\rangle \longrightarrow_{\mathsf{v}} \mathsf{F}}\ (\textsc{exh-fail})$$

$$\frac{\sigma_v \vdash \langle A, \sigma_v\rangle \longrightarrow_{\mathsf{rc}} \mathsf{N}(\sigma_v')}{\Gamma_v \vdash \langle \mathtt{assert}\ A, \sigma_v\rangle \longrightarrow_{\mathsf{v}} \mathsf{N}(\sigma_v)}\ (\textsc{assert-succ}) \qquad \frac{\sigma_v \vdash \langle A, \sigma_v\rangle \longrightarrow_{\mathsf{rc}} \mathsf{F}}{\Gamma_v \vdash \langle \mathtt{assert}\ A, \sigma_v\rangle \longrightarrow_{\mathsf{v}} \mathsf{F}}\ (\textsc{assert-fail})$$

$$\frac{\begin{array}{c}\Gamma_v \vdash \langle s_1, \sigma_v\rangle \longrightarrow_{\mathsf{v}} \mathsf{N}(\sigma_v') \\ \Gamma_v \vdash \langle s_2, \sigma_v'\rangle \longrightarrow_{\mathsf{v}} r_v\end{array}}{\Gamma_v \vdash \langle s_1; s_2, \sigma_v\rangle \longrightarrow_{\mathsf{v}} r_v}\ (\textsc{seq-n}) \qquad \frac{\Gamma_v \vdash \langle s_1, \sigma_v\rangle \longrightarrow_{\mathsf{v}} r_v \quad r_v = \mathsf{M} \vee r_v = \mathsf{F}}{\Gamma_v \vdash \langle s_1; s_2, \sigma_v\rangle \longrightarrow_{\mathsf{v}} r_v}\ (\textsc{seq-fm})$$

$$\frac{\begin{array}{c}\sigma_v \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(\mathsf{VBoolVal}(\mathsf{true})) \\ \Gamma_v \vdash \langle s_1, \sigma_v\rangle \longrightarrow_{\mathsf{v}} r_v\end{array}}{\Gamma_v \vdash \langle \mathtt{if}\ (e)\ \{s_1\}\ \mathtt{else}\ \{s_2\}, \sigma_v\rangle \longrightarrow_{\mathsf{v}} r_v}\ (\textsc{if-thn}) \qquad \frac{\begin{array}{c}\sigma_v \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(\mathsf{VBoolVal}(\mathsf{false})) \\ \Gamma_v \vdash \langle s_2, \sigma_v\rangle \longrightarrow_{\mathsf{v}} r_v\end{array}}{\Gamma_v \vdash \langle \mathtt{if}\ (e)\ \{s_1\}\ \mathtt{else}\ \{s_2\}, \sigma_v\rangle \longrightarrow_{\mathsf{v}} r_v}\ (\textsc{if-els})$$

$$\frac{\begin{array}{c}\mathsf{typVpr}(v) = \tau \\ \mathsf{UpdVars}(\Gamma_v, x, \tau) \vdash \langle s, \sigma_v(x \mapsto v)\rangle \longrightarrow_{\mathsf{v}} r_v' \\ (r_v' \in \{\mathsf{F}, \mathsf{M}\} \Rightarrow r_v'' = r_v') \wedge \forall \sigma_v'.\ r_v' = \mathsf{N}(\sigma_v') \Rightarrow r_v'' = \mathsf{N}(\sigma_v'(x \mapsto \sigma_v(x)))\end{array}}{\Gamma_v \vdash \langle \mathtt{var}\ x : \tau\ ;\ s, \sigma_v\rangle \longrightarrow_{\mathsf{v}} r_v''}\ (\textsc{scope})$$

$$\frac{\begin{array}{c}\mathsf{Methods}(\Gamma_v)(m) = mdecl \\ \sigma_v \vdash \langle es, \sigma_v\rangle\ [\Downarrow]_{\mathsf{v}}\ vs \quad \mathsf{map}(\lambda v.\ \mathsf{typVpr}(v), vs) = \mathsf{argTypes}(mdecl) \\ \mathsf{map}(\lambda z.\mathsf{Vars}(\Gamma_v)(z), zs) = \mathsf{retTypes}(mdecl) \quad \mathsf{map}(\lambda v.\ \mathsf{typVpr}(v), vs') = \mathsf{retTypes}(mdecl) \\ \Gamma_v \vdash \langle \mathtt{exhale}\ \mathsf{pre}(mdecl), (xs\ [\mapsto]\ vs, \mathsf{H}(\sigma_v), \Pi(\sigma_v))\rangle \longrightarrow_{\mathsf{v}} r_v^{pre} \\ r_v^{pre} \in \{\mathsf{F}, \mathsf{M}\} \Longrightarrow r_v = r_v^{pre} \\ \forall \sigma_v^{pre}. \left(\begin{array}{l} r_v^{pre} = \mathsf{N}(\sigma_v^{pre}) \Longrightarrow \\ \Gamma_v \vdash \langle \mathtt{inhale}\ \mathsf{post}(mdecl), (xs@ys\ [\mapsto]\ vs@vs', \mathsf{H}(\sigma_v^{pre}), \Pi(\sigma_v^{pre}))\rangle \longrightarrow_{\mathsf{v}} r_v^{post} \wedge \\ r_v = \mathsf{resetStoreAfterCall}(\mathsf{ST}(\sigma_v), zs, vs', r_v^{post}) \end{array}\right)\end{array}}{\Gamma_v \vdash \langle zs := m(es), \sigma_v\rangle \longrightarrow_{\mathsf{v}} r_v}\ (\textsc{stmt-mcall})$$

$$\frac{\begin{array}{c}es = \mathsf{defineSubExprsStmt}(s) \quad es \neq [] \\ \sigma_v \vdash \langle es, \sigma_v\rangle\ [\Downarrow]_{\mathsf{v}}\ \lightning\end{array}}{\Gamma_v \vdash \langle s, \sigma_v\rangle \longrightarrow_{\mathsf{v}} \mathsf{F}}\ (\textsc{stmt-subexp-fail})$$

Figure 3.5: Viper statement reduction rules. In STMT-MCALL, *xs* and *ys* denote the formal arguments and target variables of *m*. The term *xs* [↦] *vs* maps the *i*-th variable in the list *xs* to the *i*-th value in the list *vs*. The term *xs*@*ys* appends list *ys* to list *xs*. nonDet, defineSubExprsStmt, and resetStoreAfterCall are defined in Figure 3.6.

$$\begin{aligned}
&\mathsf{nonDet}(\sigma_v, \sigma'_v, \sigma''_v) \triangleq \quad \mathsf{ST}(\sigma''_v) = \mathsf{ST}(\sigma'_v) \wedge \pi(\sigma''_v) = \pi(\sigma'_v) \wedge \\
&\qquad \forall l.\ (\pi(\sigma_v)(l) = 0 \vee \pi(\sigma'_v)(l) > 0) \Rightarrow h(\sigma''_v)(l) = h(\sigma'_v)(l) \\
&\mathsf{defineSubExprsStmt}(s) \triangleq \quad \mathsf{case}\ s\ \mathsf{of} \\
&\qquad x := e \Rightarrow [e] \\
&\qquad \mid e_r.f := e \Rightarrow [e_r, e] \\
&\qquad \mid zs := m(es) \Rightarrow es \\
&\qquad \mid m(es) \Rightarrow es \\
&\qquad \mid \mathbf{if}\ (e)\ \{s_1\}\ \mathbf{else}\ \{s_2\} \Rightarrow [e] \\
&\qquad \mid \_ \Rightarrow [] \\
&\mathsf{defineSubExprsA}(A) \triangleq \quad \mathsf{case}\ A\ \mathsf{of} \\
&\qquad e \Rightarrow [e] \\
&\qquad \mid \mathbf{acc}(e.f, e_p) \Rightarrow [e, e_p] \\
&\qquad \mid e \Rightarrow A \Rightarrow [e] \\
&\qquad \mid e\ ?\ A : B \Rightarrow [e] \\
&\qquad \mid \_ \Rightarrow [] \\
&\mathsf{resetStoreAfterCall}(st, zs, vs', r_v) \triangleq \quad \mathsf{case}\ r_v\ \mathsf{of} \\
&\qquad \mathsf{F} \Rightarrow \mathsf{F} \\
&\qquad \mid \mathsf{M} \Rightarrow \mathsf{M} \\
&\qquad \mid \mathsf{N}(\sigma_v) \Rightarrow \mathsf{N}((st(zs[\mapsto]vs'), \mathsf{H}(\sigma_v), \Pi(\sigma_v)))
\end{aligned}$$

Figure 3.6: Auxiliary definitions for the semantics. The term $st(zs[\mapsto]vs')$ denotes the store st except for variables zs, which are mapped to values vs' (i-th variable in zs is mapped to i-th variable in vs').

Recall that, in our semantics, the heap in a Viper state is a *total* mapping from heap locations to values. It is crucial that whenever an `inhale` obtains permission to a heap location `x.f` for which there was no permission before, then the semantics takes every possible value for `x.f` into account. Our semantics achieves this by considering every possible heap at the beginning of a program execution, and nondeterministically choosing values for those heap locations for which an `exhale` removes all available permission. This choice precisely reflects the Viper semantics and is motivated by the existing Viper-to-Boogie implementation, which uses the same high-level approach in the Boogie encoding. We discuss this design choice in Subsection 3.2.8.

The decomposition of the assertion from left to right directly reflects how Viper treats permission introspection in separating conjunctions. This shows another instance of the impact permission introspection has on the formalisation, and thus motivates including permission introspection in our semantics in order to make our certification work extensible to larger Viper subsets. For instance, executing `inhale acc(x.f, write ) && perm(x.f) == none` in a state without any permissions always goes to magic (`perm(x.f) == none` evaluates to false), because permission introspection in the right conjunct, within an `inhale`, refers to the permission *after* inhaling the left conjunct. Similarly, `exhale acc(x.f, write) && perm(x.f) == none` succeeds in a state with full permission to `x.f` (`perm(x.f) == none` evaluates to true), because permission introspection in the right conjunct, within an `exhale`, refers to the permission *after* removing the permission to the left conjunct.

Formalising this decomposition is straightforward for `inhale`. We express the semantics of `inhale` A && B as the execution of `inhale` A followed by `inhale` B. In addition to handling permission introspection as expected,

$$\frac{\begin{array}{c}\sigma_v \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{VBoolVal}(b)\\ r_v = \textit{if } b \textit{ then } \mathsf{N}(\sigma_v) \textit{ else } \mathsf{M}\end{array}}{\langle e, \sigma_v\rangle \longrightarrow_{\mathsf{inh}} r_v}\ (\textsc{inh-exp}) \qquad \frac{\begin{array}{c} es = \mathsf{defineSubExprsA}(A) \quad es \neq [\,]\\ \sigma_v \vdash \langle es, \sigma_v\rangle\ [\Downarrow]_{\mathsf{v}}\ \lightning\end{array}}{\langle A, \sigma_v\rangle \longrightarrow_{\mathsf{inh}} \mathsf{F}}\ (\textsc{inh-subexp-fail})$$

$$\frac{\begin{array}{c}\langle A, \sigma_v\rangle \longrightarrow_{\mathsf{inh}} \mathsf{N}(\sigma_v')\\ \langle B, \sigma_v'\rangle \longrightarrow_{\mathsf{inh}} r_v\end{array}}{\langle A\ \&\&\ B, \sigma_v\rangle \longrightarrow_{\mathsf{inh}} r_v}\ (\textsc{inh-sep-n}) \qquad \frac{\begin{array}{c}\langle A, \sigma_v\rangle \longrightarrow_{\mathsf{inh}} r_v\\ r_v \in \{\mathsf{F}, \mathsf{M}\}\end{array}}{\langle A\ \&\&\ B, \sigma_v\rangle \longrightarrow_{\mathsf{inh}} r_v}\ (\textsc{inh-sep-fm})$$

$$\frac{\begin{array}{c}\sigma_v \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(r) \quad \sigma_v \vdash \langle e_p, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(p)\\ p < 0 \Rightarrow r_v = \mathsf{F}\\ p \geq 0 \Rightarrow r_v = \textit{if } \mathsf{inhSucc(r, p)} \textit{ then } \mathsf{N}(\sigma_v') \textit{ else } \mathsf{M}\\ \sigma_v' = \mathsf{addperm}(\sigma_v, r, f, p)\end{array}}{\langle \mathtt{acc}(e.f, e_p), \sigma_v\rangle \longrightarrow_{\mathsf{inh}} r_v}\ (\textsc{inh-acc})$$

$$\mathsf{inhSucc}(r, p) \triangleq (p > 0 \Rightarrow r \neq \mathtt{null}) \land (r \neq \mathtt{null} \Rightarrow p + \Pi(\sigma_v)(r, f) \leq 1)$$

Figure 3.7: Selected rules for the formal semantics of `inhale`. $\mathsf{addperm}(\sigma_v, r, f, p)$ denotes the state σ_v where permission p has been added to (r, f). $\mathsf{defineSubExprsA}(A)$ is defined in Figure 3.6.

this semantics also correctly captures the evaluation of field accesses in assertions. For instance, when executing `inhale acc(x.f, write) && x.f > 0`, the field access `x.f` in the second conjunct is evaluated in the state *after* inhaling full permission to `x.f`. This is the expected semantics: in the example, the evaluation of `x.f` is always well-defined.

Formalising the decomposition for `exhale` A is more complex. The reason is that, contrary to `inhale`, field accesses in A must be evaluated in the state *before* the `exhale`. For instance, in `exhale acc(x.f, write) && x .f > 0`, `x.f` in the second conjunct must be evaluated in the state before the `exhale`. Evaluating `x.f` after executing `exhale acc(x.f, write)` would be ill-defined; the heap would store some nondeterministic value for `x.f`. Our semantics for `exhale` A handles this by first removing the permissions and checking the constraints specified in A *without changing the heap yet* via an intermediate operation `remcheck` A; only after this intermediate operation does our semantics apply nondeterministic assignments. We then express the semantics of `remcheck` A && B by executing `remcheck` A followed by executing `remcheck` B.

The semantics of `remcheck` must take two different states into account when evaluating expressions: the state before the `exhale`, which we call the *permission definedness state*, and the state being updated during the `remcheck`, which we call the *reduction state*. This allows a correct treatment of field accesses and permission introspection: the former state is used to check whether field accesses are well-defined and the latter is used for the remaining cases including the evaluation of permission introspection. The necessity for these two states motivates having two states in the expression evaluation judgement.

Let us now take closer look at the judgement formalising `inhale`. The reduction of `inhale` A for an assertion A from state σ_v to outcome r_v is expressed via the judgement $\langle A, \sigma_v\rangle \longrightarrow_{\mathsf{inh}} r_v$ (see rule INH in Figure 3.5 on page 113), which is inductively defined via the rules shown in Figure 3.7. Our semantics for $\langle A, \sigma_v\rangle \longrightarrow_{\mathsf{inh}} r_v$ decomposes A from left

$$\frac{\begin{array}{c}\sigma_v \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{VBoolVal}(b) \\ r_v = \mathit{if}\ b\ \mathit{then}\ \mathsf{N}(\sigma_v)\ \mathit{else}\ \mathsf{F}\end{array}}{\sigma_v^0 \vdash \langle e, \sigma_v\rangle \rightarrow_{\mathsf{rc}} r_v}\ (\textsc{rc-exp}) \qquad \frac{\begin{array}{c}\sigma_v^0 \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(r) \quad \sigma_v^0 \vdash \langle e_p, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(p) \\ r_v = \mathit{if}\ \mathsf{exhAccSucc}(r, p, \sigma_v)\ \mathit{then}\ \mathsf{N}(\sigma_v^R)\ \mathit{else}\ \mathsf{F}\end{array}}{\sigma_v^0 \vdash \langle \mathbf{acc}(e.f, e_p), \sigma_v\rangle \rightarrow_{\mathsf{rc}} r_v}\ (\textsc{rc-acc})$$

$$\frac{\begin{array}{c}\sigma_v^0 \vdash \langle A, \sigma_v\rangle \rightarrow_{\mathsf{rc}} \mathsf{N}(\sigma_v') \\ \sigma_v^0 \vdash \langle B, \sigma_v'\rangle \rightarrow_{\mathsf{rc}} r_v\end{array}}{\sigma_v^0 \vdash \langle A \mathbin{\&\&} B, \sigma_v\rangle \rightarrow_{\mathsf{rc}} r_v}\ (\textsc{rc-sep-n}) \qquad \frac{\sigma_v^0 \vdash \langle A, \sigma_v\rangle \rightarrow_{\mathsf{rc}} \mathsf{F}}{\sigma_v^0 \vdash \langle A \mathbin{\&\&} B, \sigma_v\rangle \rightarrow_{\mathsf{rc}} \mathsf{F}}\ (\textsc{rc-sep-f})$$

$$\frac{\begin{array}{c}\sigma_v^0 \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(\mathsf{VBoolVal}(\mathsf{true})) \\ \sigma_v^0 \vdash \langle A, \sigma_v\rangle \rightarrow_{\mathsf{rc}} r_v\end{array}}{\sigma_v^0 \vdash \langle e \Rightarrow A, \sigma_v\rangle \rightarrow_{\mathsf{rc}} r_v}\ (\textsc{rc-impT}) \qquad \frac{\sigma_v^0 \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(\mathsf{VBoolVal}(\mathsf{false}))}{\sigma_v^0 \vdash \langle e \Rightarrow B, \sigma_v\rangle \rightarrow_{\mathsf{rc}} \mathsf{N}(\sigma_v)}\ (\textsc{rc-impF})$$

$$\frac{\begin{array}{c}\sigma_v^0 \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(\mathsf{VBoolVal}(\mathsf{true})) \\ \sigma_v^0 \vdash \langle A, \sigma_v\rangle \rightarrow_{\mathsf{rc}} r_v\end{array}}{\sigma_v^0 \vdash \langle e\ ?\ A : B, \sigma_v\rangle \rightarrow_{\mathsf{rc}} r_v}\ (\textsc{rc-condT}) \qquad \frac{\begin{array}{c}\sigma_v^0 \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(\mathsf{VBoolVal}(\mathsf{false})) \\ \sigma_v^0 \vdash \langle B, \sigma_v\rangle \rightarrow_{\mathsf{rc}} r_v\end{array}}{\sigma_v^0 \vdash \langle e\ ?\ A : B, \sigma_v\rangle \rightarrow_{\mathsf{rc}} r_v}\ (\textsc{rc-condF})$$

$$\frac{\begin{array}{c}es = \mathsf{defineSubExprsA}(A) \quad es \neq [] \\ \sigma_v \vdash \langle es, \sigma_v\rangle\ [\Downarrow]_{\mathsf{v}}\ \lightning\end{array}}{\sigma_v^0 \vdash \langle A, \sigma_v\rangle \rightarrow_{\mathsf{rc}} \mathsf{F}}\ (\textsc{rc-subexp-fail})$$

$$\mathsf{exhAccSucc}(r, p, \sigma_v) \triangleq p \geq 0 \wedge (\mathit{if}\ r = \mathbf{null}\ \mathit{then}\ p = 0\ \mathit{else}\ \Pi(\sigma_v)(r, f) \geq p)$$

$$\sigma_v^R \triangleq \mathsf{rem}(\sigma_v, r, f, p)$$

Figure 3.8: The rules for the formal semantics of **remcheck**. $\mathsf{rem}(\sigma_v, r, f, p)$ is the state σ_v where permission p is removed from (r, f). $\mathsf{defineSubExprsA}(A)$ is defined in Figure 3.6.

to right as shown by the rules INH-SEP-N and INH-SEP-FM for the separating conjunction: that is, for an assertion A && B, first A is inhaled, and if this succeeds, then B is inhaled, otherwise the outcome is given by the outcome of inhaling A.

The **inhale** accessibility predicate rule INH-ACC in Figure 3.7 expresses that if the added permission is negative then the operation fails. If the permission is nonnegative, then the operation succeeds if (1) the receiver is non-null if $p > 0$, and (2) the added permission does not yield an inconsistent state (*i.e.* does not result in more than full permission for heap location (r, f)). Otherwise, the operation goes to magic (denoted by outcome M). If the operation succeeds (*i.e.* results in a normal outcome), then the new state additionally contains the added permission p at heap location (r, f).

The **inhale** expression rule INH-EXP in Figure 3.7 reflects that if the expression does not hold, then the **inhale** goes to magic, and otherwise the **inhale** has no effect. This case is analogous to the semantics for the **assume** command in Boogie (except that the Viper expression can be ill-defined, which results in **inhale** failing).[10]

10: Viper also supports a separate assume primitive. In the case when a Boolean expression is assumed, the assume primitive has a meaning identical to **inhale**. We currently do not model assume primitives with general assertions (*e.g.* with accessibility predicates); defining the corresponding semantics would be straightforward.

Let us now discuss the judgements formalising **exhale**. The inference rule EXH-SUCC in Figure 3.5 on page 113 formalises the behaviour for the case when **exhale** A succeeds. The big-step judgement $\sigma_v \vdash \langle A, \sigma_v\rangle \rightarrow_{\mathsf{rc}} \mathsf{N}(\sigma_v')$ defines the successful execution of a **remcheck** A operation from σ_v to σ_v' (removing the permissions while leaving the heap unchanged). nonDet specifies the nondeterministic assignment for all heap locations for which **remcheck** A removed all permission. The case when **remcheck** A (and thus **exhale** A) fails, is captured by the rule EXH-FAIL in Figure 3.5.

The rules formalising **remcheck** are shown in Figure 3.8. Our semantics for **remcheck** A decomposes the assertion A from left to right analogously to our semantics for **inhale** A. As discussed, the semantics of **remcheck** must take two states into account: Our judgement carries both a *permission definedness state* (σ_v^0 in rule RC-SEP-N) in which permissions for field accesses are checked and a *reduction state* (σ_v and σ_v' in rule RC-SEP-N) from which permissions are removed. For each of the rules defining $\sigma_v^0 \vdash \langle A, \sigma_v \rangle \longrightarrow_{\mathsf{rc}} r_v$ (Figure 3.8), if a subexpression e of A is evaluated in the rule's premise, then it is done so using the judgement $\sigma_v^0 \vdash \langle e, \sigma_v \rangle \Downarrow_{\mathsf{V}} r_v$. This accurately reflects the evaluation of field accesses and permission introspection: field accesses are checked to be well-defined in σ_v^0 and permission introspection looks up the permissions in σ_v.

Note that the permission definedness state and the reduction state differ at most on the permission mask. So, if e has no permission introspection, then $\sigma_v^0 \vdash \langle e, \sigma_v \rangle \Downarrow_{\mathsf{V}} r_v$ is equivalent to $\sigma_v^0 \vdash \langle e, \sigma_v^0 \rangle \Downarrow_{\mathsf{V}} r_v$.[11] Also note that in every rule in our Viper semantics for constructs other than **exhale** (and **remcheck**), the two states in the expression evaluation judgements are always the same. This illustrates that the expression evaluation judgement needs to take two states as parameters only because of the potential combination of permission introspection and **exhale**, and thus shows an instance of the impact permission introspection has on the formalisation.

11: This usage of a single state instead of two states for the expression evaluation is precisely what we did in the presentation of the accompanying conference paper [60], since we did not include permission introspection in the paper.

Rule RC-ACC in Figure 3.8 for **remcheck** **acc**$(e.f, e_p)$ models removing e_p permission from heap location $e.f$. The operation succeeds (expressed by $\mathsf{exhAccSucc}(r, p, \sigma_v)$) iff (1) the to-be-removed permission is nonnegative and, (2) there is sufficient permission. Rule RC-ACC is applicable only if e and e_p are well-defined, otherwise **remcheck** **acc**$(e.f, e_p)$ fails as captured by rule RC-SUBEXP-FAIL.

Rule RC-EXP in Figure 3.8 reflects that **remcheck** e fails if the Boolean expression e does not hold, and otherwise has no effect. This behaviour is identical to the semantics for **assert** e in Viper and analogous to the **assert** command in Boogie (the main difference to Boogie is that Boogie expressions are never ill-defined). As discussed, **assert** A in Viper for a general assertion A differs from **exhale** A. Executing **assert** A checks whether **remcheck** A succeeds, and if it does, then **assert** A results in a normal outcome whose state is the same as before the execution of **assert** A, and otherwise fails if **remcheck** A fails (see rules ASSERT-SUCC and ASSERT-FAIL in Figure 3.5).

Method calls

Finally, let us consider the semantics of a method call $m(es)$ to callee m (*i.e.* m does not return any results in this case). Intuitively, the semantics of such a call is to ensure that m's precondition holds and to remove the corresponding permissions before transitioning to a state in which the postcondition holds and the corresponding permissions are added. If the precondition does not hold (*e.g.* the caller does not have the specified permission), then the call fails. In addition to specifying permissions, the pre- and postcondition can constrain heap locations (*e.g.* the postcondition may specify how the method modified a heap location). If the caller has p permission to `x.f` but m's precondition specifies *less than* p permission to `x.f` (or potentially does not specify any permission to `x.f` at all), then the call's semantics makes sure that the corresponding value of `x.f` remains unchanged by the call. In separation logic terminology, in this case, the heap information and permission to `x.f` is *framed* around the call.

The semantics of a method call $zs := m(es)$, where the call returns results that are stored in target variables zs, is almost identical to the semantics when a call does not return any results. The main difference is that the postcondition may additionally constrain the returned results. So, the semantics additionally takes into account every possible returned result that satisfies the postcondition.

The rule STMT-MCALL in Figure 3.5 on page 113 formalises the semantics of method calls. (If the target variables zs in the rule are empty, then one obtains the semantics of a call that does not return results.) STMT-MCALL expresses the semantics by (1) exhaling the precondition (*i.e.* ensuring the precondition holds and removing the specified permissions), (2) nondeterministically assigning values vs' of the declared type to the target variables, and (3) inhaling the postcondition (*i.e.* transitioning to a state in which the postcondition holds and adding the specified permissions). Note that this semantics ensures that if the caller has permissions to a heap location that are not specified by the precondition, then the heap values remain the same. This is because in such case the **`exhale`** would not change such a heap location, since there would be still be permission left to this heap location after the **`exhale`**. Moreover, the semantics ensures that all information is lost on heap locations after the **`exhale`** for which the precondition specifies all the permissions held by the caller. This models the fact that the call could change those heap locations, and thus, the caller should obtain the information on those locations only via the postcondition.

To provide an intuition for the method call semantics, consider the following method declaration:

```
method setPositive(x: Ref) returns (z: Int)
  requires acc(x.f, write)
  ensures  acc(x.f, write) && x.f == z && z > 0
```

The precondition requires full permission to `x.f`. The postcondition ensures the same permission and also ensures that `x.f` evaluates to the returned value `z` and that this value is positive. Now, consider a call `i := setPositive(r)`. This call should succeed only if there is full permission to `r.f` before the call, which is ensured by the initial **`exhale`** operation of the precondition in the call's semantics. If the **`exhale`** succeeds, then the subsequent nondeterministic assignment to `i` and the **`inhale`** of the postcondition will ensure that the caller will get back permission to `r.f` and after the call `r.f` stores a positive value given by the value returned by the call. The successful **`exhale`** of the precondition ensures that all information on `r.f` is lost (since full permission is exhaled, the **`exhale`** nondeterministically assigns a value to `r.f`). Otherwise, if `r.f` was negative before, the **`inhale`** could yield contradicting information and result in a magic state, which would not accurately capture the call.

Note that m's pre- and postcondition appear in STMT-MCALL precisely as they do in the signature of m, that is, with the *formal arguments xs* and the *formal target variables ys*. As a result, the rule expresses the **`exhale`** and **`inhale`** operations in states whose stores contain only the formal arguments and formal target variables. If the **`exhale`** and **`inhale`** operations succeed, then the store is reset to the store before the call and where the target variables are updated accordingly (via resetStoreAfterCall). If the operations do not succeed, then the call's outcome is a failure or magic outcome (*e.g.* if either operation fails, then the call fails). Finally, note that the call reduces only if the actual argument expressions *es* and the actual target variables *zs* respect the types in the callee's signature, which keeps states well-typed as in the variable and field assignment rules; this has no effect on well-typed programs.

Expressing the method call rule via substitution

An alternative approach to expressing the method call rule would be to consider the callee's pre- and postcondition where the formal arguments and formal target variables are substituted by the actual arguments and actual target variables. One reason why we did not use this approach is that if the actual arguments have *old expressions* (which the Viper subset presented in this dissertation does not include), then the substitution would not reflect the semantics accurately. In particular, the old expressions in the actual arguments (referring to the pre-state of the caller) would be conflated with old expressions in the declared postcondition (referring to the pre-state of the callee). One could circumvent this issue, for example, by renaming the old expressions in the actual arguments to *labelled old expressions* (also not included in the subset here) using a fresh label and adjusting the Viper state accordingly.

3.2.6. Correctness of a Viper Program

A Viper program is correct if each of its methods is correct. The correctness of a method m w.r.t. field declarations F and method declarations M is given by the following definition:

Definition 3.2.1 (Correctness of a Viper method)

$$\begin{aligned}
&\mathit{methodCorrect}^{F,M}(m) \triangleq\\
&\quad \forall \sigma_v.\ [\mathit{stateWellTyVpr}(F, m, \sigma_v) \wedge (\forall l.\ \Pi(\sigma_v)(l) = 0)] \Longrightarrow\\
&\qquad \text{let } s_v = \textbf{inhale}\ \mathit{pre}(m); \mathit{body}(m); \textbf{exhale}\ \mathit{post}(m) \text{ in}\\
&\qquad \forall r_v.\ \mathit{initCtxt}_v^{F,M}(m) \vdash \langle s_v, \sigma_v\rangle \longrightarrow_{\mathsf{V}} r_v \Longrightarrow r_v \neq \mathsf{F}
\end{aligned}$$

where stateWellTyVpr(F, m, σ_v) *expresses that* σ_v *is well-typed (i.e. the store respects the declarations of formal arguments and target variables in method* m*, and the heap respects the field declarations* F*), and* initCtxt$_v^{F,M}(m)$ *constructs the initial Viper context.*

Intuitively, m is correct, if for any initial state that satisfies m's precondition, any execution of m's body in this state must result in a magic outcome or in a normal outcome with a state that satisfies m's postcondition. In our semantics, we model the satisfaction of assertions via **inhale** and **exhale**. The normal outcomes resulting from inhaling assertion A from all well-typed states yields the set of states that satisfy A. If exhaling A from a state cannot fail, then the state satisfies A. As a result, it is natural to express the correctness of a Viper method by requiring that any execution starting in a well-typed state σ_v that inhales the precondition, then executes the body, and finally exhales the postcondition, cannot fail. For our supported Viper subset, such a definition would provide the expected results for method bodies that do *not* use permission introspection.

Definition 3.2.1 defines the correctness of a Viper method; this definition also works with permission introspection. The only difference to the definition that we just described is that Definition 3.2.1 considers only those executions that inhale the precondition from well-typed states *without any permissions* (*i.e.* $(\forall l.\ \Pi(\sigma_v)(l) = 0)$). For instance, the following method is correct in Viper:

```
method m(x: Ref)
  requires acc(x.f, 1/2)
  ensures  true
{
  assert perm(x.f) == 1/2
}
```

Inhaling `acc(x.f, 1/2)` in a state without any permission always yields a state with exactly half permission. As a result, the **assert** statement in the method body always succeeds. If one removed the empty permission restriction, then the **assert** statement would fail, since executions with more than half permission would reach the **assert** statement. This observation shows another instance of the impact that permission introspection has on the Viper semantics. In our supported Viper subset, one can prove that if the method body has no permission introspection, then the empty permission restriction does not affect the correctness of a method.

Treatment of method calls

Note that Definition 3.2.1 accurately captures the correctness of all possible Viper methods in our subset, including recursive methods. In particular, when considering the correctness of a method m w.r.t. field declarations F and method declarations M (denoted by $\mathsf{methodCorrect}^{F,M}(m)$), M contains *all* methods in the corresponding Viper program including m. Since our semantics for a call to a method m just uses m's specification, there are only finite executions, and the correctness of a method does not depend on the correctness of any other method. Since Viper is primarily an intermediate verification language, such a correctness definition is often sufficient. In particular, executions of a Viper method often do not directly reflect executions in an input program that is translated to Viper (*e.g.*, concurrent input programs are translated into multiple sequential Viper methods). Instead, when reasoning about such translations from some input language into Viper, one would show that a modular treatment of calls via specifications at the Viper level implies a correctness result on the input program w.r.t. an input language semantics where the *bodies* of callees in the input program are executed.

Nevertheless, there are cases where it is useful to know that an execution of a method m satisfies m's specification and where the semantics of calls executes the callee's body (for instance, when Viper is not used as an intermediate verification language). In such a case, one would have to define an alternative semantics that treats calls differently. One could then prove that our notion of correctness implies an alternative notion of correctness that uses such an alternative semantics (potentially under certain conditions, for example, since permission introspection may lead to incomparable executions).

If one wanted to also ensure that there are no nonterminating executions, then one would have to ensure that an alternative semantics is able to express nontermination using, for example, a *small-step* semantics or a *coinductive* big-step semantics (an *inductive* big-step semantics as used in our formalisation cannot express nontermination). In this case, our correctness definition (Definition 3.2.1) would not imply termination in general. Instead, one would require some extra condition that, for instance, expresses a decreasing and terminating measure on method calls. Viper supports the specification of such measures and Viper verifiers also prove termination w.r.t. a semantics that executes the bodies of method calls using such measures; we do not consider such measures in this dissertation.

3.2.7. Illustrating the Viper Semantics on an Example

We will now discuss at a high level why the Viper program shown in Figure 3.2 on page 107 is correct, which we introduced in Subsection 3.2.2. This discussion will give an intuition for our formally introduced Viper semantics.

To show that this program is correct, we need to show that its methods `main` and `sum` are correct. Let us first see why method `main` is correct. Since `main`'s pre- and postcondition are trivial (*i.e.* **`true`**), `main` is correct (Definition 3.2.1) iff `main`'s body has no failing executions starting from any well-typed state with no permissions. The first two inhales in `main`'s body clearly do not fail, since the corresponding receivers are trivially well-defined (they are both variables) and the permission amounts are nonnegative. The inhales lead to a normal outcome and not to a magic outcome for every considered execution, since heap locations with different fields are always different. Thus, the resulting permission mask is always consistent here. The two assignments succeed, because the two inhales guarantee full permission to both target heap locations. The semantics of the call to `sum` first exhales `sum`'s precondition and then inhales `sum`'s postcondition. The **`exhale`** succeeds since (1) full permission to both `x.f` and `y.g` is still held, and (2) the logical constraints in `sum`'s precondition hold. The **`inhale`** also succeeds and ensures that after the call `main` still has full permission to both `x.f` and `y.g`, and that `x.f` equals `(y.g * (y.g + 1)) / 2`.

The most interesting aspect of `main`'s correctness is arguing why the final **`assert`** succeeds in every execution that reaches it. The fact that in every execution that reaches the **`assert`** statement `x.f` equals `(y.g * (y.g + 1)) / 2` is not sufficient on its own to deduce success. We additionally need that `y.g` is 100, that is, `y.g` has the same value as *before* the call to `sum`. This is indeed the case, because the precondition of `sum` specifies only *half* permission to `y.g`. As a result, the **`exhale`** of the precondition leaves the value stored in the heap for `y.g` unchanged, since after the **`exhale`** there is still half permission left. Therefore, `y.g`'s value is indeed 100. In separation logic terminology, the heap information of `y.g` is *framed* around the call, since the call does not require all of the available permission to `y.g`.

Let us now see why method `sum` is correct. The precondition of `sum` states that before the call `x.f` must be equal to the sum of all integers between 1 and `i-1` and that `i` is at most `y.g + 1`. The postcondition of `sum` states that after the call `x.f` must be equal to the sum of all integers between 1 and `y.g`. (The specification of `sum` expresses the sum of all integers between 1 and k using the well-known formula $\frac{k*(k+1)}{2}$.) Intuitively, this postcondition must hold given the precondition, since `sum` adds the sum of all integers between `i` and `y.g` to `x.f`.

To show that `sum` is correct, one must show that starting from any well-typed state with no permissions, inhaling the precondition, then executing the body, and finally exhaling the postcondition never fails. The **`inhale`** of the precondition succeeds, since for each heap location evaluated in the precondition, the corresponding permission is inhaled first. Using similar arguments as for `main` it is straightforward to see why executing `sum`'s body after the **`inhale`** never fails. Finally, the **`exhale`** of the postcondition does not fail at the end because of the following. If `i` is larger than `y.g`, then no further statements are executed in the body. In this case, the precondition specifies all the permissions and logical constraints specified in the postcondition, since from `i <= y.g + 1` we learn that `i` is equal to `y.g + 1`. Thus, the **`inhale`** of precondition adds all the permissions and assumes all constraints required for the **`exhale`** to not fail. If `i` is at most `y.g`, then the then-branch of the body is executed. Here, the postcondition inhaled as part of the call to `sum` precisely matches the postcondition that needs to be exhaled for the correctness. Thus, the **`exhale`** does not fail.

3.2.8. Semantics Design Decisions

The semantics that we have defined in this section accurately reflects Viper for the supported subset. There are various ways one could have formalised different parts of the semantics. In this subsection, we discuss two of our design decisions.

A first design decision is that we use an *operational* Viper semantics instead of, for example, an axiomatic semantics. There are two motivations for this design choice. First, an operational Viper semantics simplifies the formal connection to an operational Boogie semantics of Boogie programs generated by the existing Viper-to-Boogie translation, which is the main goal of this chapter. The operational semantics of Viper statements is described via a sequence of state updates and checks. These sequences of state updates and checks connect naturally to a simulating sequence of Boogie state updates and checks. Moreover, our semantics for **`inhale`** A and **`exhale`** A is itself described operationally: both operations decompose A from left to right, updating the state along the way, which is also how the existing Viper-to-Boogie translation encodes these statements into Boogie. Second, as we discuss in Subsection 3.2.5 on page 112, permission introspection requires decomposing the assertion from left to right for `inhale` and `exhale`, which naturally leads to an operational description of both operations. Since one goal of this chapter is to provide solutions that are extensible to larger Viper subsets, it was important to take permission introspection into account. We discuss a Viper semantics not presented in this dissertation as part of related work (Section 3.7), where we do not support permission introspection, and where we instead use an axiomatic semantics, and a more abstract description of `inhale` and `exhale`.

A second design decision is that we use a total mapping for the heap instead of a partial mapping. This is a common choice for implicit dynamic frames, but in our case the main motivation was that the existing Viper-to-Boogie translation encodes the heap as a total heap in Boogie. As a result, using a total mapping in the Viper semantics makes the connection to Boogie simpler. Moreover, our semantics nondeterministically assigns values to heap locations for which all permissions are lost as part of an `exhale`. However, when inhaling permission to a heap location without any permission, the corresponding value is not nondeterministically assigned. This setup also matches how the existing Viper-to-Boogie translation encodes `exhale` and `inhale`, and thus simplifies the formal validation. However, this setup is slightly unintuitive, because the property one needs is that whenever an `inhale` operation obtains permission to a heap location for which there was no permission before, then the semantics must consider every possible heap value for this location. In work not presented in this dissertation [45], we formally show that this property indeed holds by connecting our semantics to a Viper semantics with partial heaps, where heap values are assigned nondeterministically when inhaling.

[45]: Dardinier et al. (2025), *Formal Foundations for Translational Separation Logic Verifiers*

This concludes our presentation of our formal semantics for a subset of Viper. Our automatically generated certificates establish soundness of the existing Viper-to-Boogie translation w.r.t. this semantics. We will next take a closer look at the existing Viper-to-Boogie translation, followed by a presentation of our general methodology for formally validating front-end translations (Section 3.4), which we then apply to the existing Viper-to-Boogie translation (Section 3.5).

3.3. The Existing Viper-to-Boogie Translation

In this section, we present aspects of the existing Viper-to-Boogie translation relevant for this chapter. First, we provide concrete examples taken from the existing Viper-to-Boogie translation to showcase the three challenges that arise when formally validating front-end translations discussed at a high level in Section 3.1: the semantic gap (Subsection 3.3.1), diverse translations (Subsection 3.3.2), and non-locality (Subsection 3.3.3). Next, we provide more details on the non-local check performed by the existing Viper-to-Boogie translation, including why the check justifies the corresponding optimisation (Subsection 3.3.4). Then, we discuss the Boogie features that we additionally formalised to capture Boogie programs targeted by the existing Viper-to-Boogie translation and which

```
inhale acc(x.f, q)
y.g := x.f+1
exhale acc(x.f, q) && y.g > x.f
```

↝

```
tmp := q;
assert tmp >= 0;
assume tmp > 0 ==> x != null;
M[x,f] += tmp;
assume GoodMask(M);
assert M[x,f] > 0;
assert M[y,g] == 1;
H[y,g] := H[x,f]+1;
assume GoodMask(M);
WM := M;
tmp := q;
assert tmp >= 0;
if(tmp != 0) {
  assert M[x,f] >= tmp;
}
M[x,f] -= tmp;
assert WM[y,g] > 0;
assert WM[x,f] > 0;
assert H[y,g] > H[x,f];
havoc H';
assume idOnPositive(H,H',M);
H := H';
assume GoodMask(M);
```

Figure 3.9: A Viper statement (on the left) and the corresponding (simplified) Boogie statement (on the right) that is emitted by the existing Viper-to-Boogie translation.

were not formalised in Chapter 2 (Subsection 3.3.5). Next, we provide a high-level overview of the Boogie background declarations generated by the existing translation and how to express the polymorphic Boogie maps generated by the existing translation in the Boogie subset formalised in Chapter 2 using background declarations (Subsection 3.3.6). Finally, we show how to use Boogie values to capture Boogie type constructors generated by the existing translation as part of the background declarations (Subsection 3.3.7). That is, we show how to instantiate the carrier type representing Boogie's *abstract* values, which represent inhabitants of types obtained via the Boogie type constructors; the carrier type is a parameter of the Boogie value definition formalised in Subsection 2.3.3 on page 27.

3.3.1. Challenge 1: The Semantic Gap

To give a flavour of the large semantic gap between a Viper statement and the corresponding Boogie translation, consider Figure 3.9, which shows a concrete Viper statement and the corresponding Boogie statement emitted by the existing Viper-to-Boogie translation (in simplified form). The Viper statement first adds permission to `x.f` via the **`inhale`** operation, then updates `y.g`, and finally removes the added permission to `x.f` and checks that `y.g` is greater than `x.f` via the **`exhale`** operation. This sequence of operations can appear in a Viper method body. Moreover, this sequence of operations essentially describes the correctness of a Viper method (Definition 3.2.1) with the permission to `x.f` as precondition, the field update as method body, and the exhaled assertion as postcondition. As a result, the existing translation of such a method also translates such a sequence.

The corresponding Boogie statement is significantly larger. The **inhale** is encoded on lines 1-5, the assignment is encoded on lines 6-9, and the **exhale** is encoded on lines 10-23. One reason for this large difference in size is that Boogie has no built-in support for implicit dynamic frames (IDF) reasoning. In particular, Boogie has no built-in heap or notion of permissions, and in contrast to Viper, Boogie's expressions never fail to evaluate. As a result, the Viper state must be represented explicitly in Boogie and the well-definedness conditions for expressions must be encoded explicitly via Boogie **assert** commands. Moreover, the nontrivial **inhale** and **exhale** operations that operate on IDF assertions must be encoded into Boogie's simple basic commands. One consequence is that the success conditions for these operations must be reflected explicitly via **assert** commands. Moreover, conditions guaranteed by these operations must be reflected explicitly via **assume** commands.

The Boogie program uses map-typed variables `H` and `M` to explicitly model the Viper heap and permission mask, respectively.[12] Permission requirements on the current permission mask are modelled via conditions on `M`. For instance, line 6 checks whether there is permission to `x.f` to ensure that the expression assigned to `y.g` is well-defined, and line 7 checks whether there is full permission to `y.g` as required by the semantics of a field assignment. Note that `M` is not always the only variable tracking permissions. For instance, in the encoding of the **remcheck** operation as part of the **exhale** statement, the auxiliary variable `WM` captures the permission mask of the corresponding permission definedness state (line 10). All heap locations evaluated during the **remcheck** operation are checked to have positive permission w.r.t. `WM`.

12: We describe in Subsection 3.3.6 how we represent such maps using the Boogie subset formalised in Chapter 2.

The Boogie program uses uninterpreted Boogie functions constrained via Boogie axioms to model different semantic notions of the Viper statement. This is another aspect of the semantic gap since these semantic notions modelled explicitly in the Boogie program via functions and axioms are implicit in the Viper program. For instance, the uninterpreted function `GoodMask` expresses when a permission mask is consistent. An axiom constrains the function correspondingly; we will show the axiom in Subsection 3.3.6. As another example, the Boogie program encodes the nondeterministic assignment of heap values at the end of the **exhale** operation using the uninterpreted function `idOnPositive` (lines 20-22). In the encoding, a heap `H'` is nondeterministically obtained via **havoc** `H'` and then constrained to match the original heap `H` on all locations where there *is* positive permission (w.r.t. `M`) via the **assume** statement. This constraint is achieved using `idOnPositive`; we will show the corresponding axiom in Subsection 3.3.6. Note that this Boogie encoding overapproximates the nondeterministic assignment specified by the Viper semantics: assigning new values to *all* locations without permission, rather than only those newly without permission.

Even the tiny snippet of Viper code shown in Figure 3.9 illustrates the explosion in concerns, complexity and the inobvious mapping between concepts in one language and the other, all of which must be taken care of in a formal validation approach.

```
aH,aM := H,M;
//remcheck A in aH,aM
...
//continue with H,M
```

```
tH,tM := H,M;
//remcheck A in H,M
...
H,M := tH,tM;
//continue with H,M
```

```
if(*) {
  //remcheck A in H,M
  ...
  assume false;
}
//continue with H,M
```

Figure 3.10: Three viable translations into Boogie for the Viper statement `assert A`. The translation on the left is the one used by the existing Viper-to-Boogie translation if `A` has accessibility predicates.

3.3.2. Challenge 2: Diverse Translations

Front-end translations implemented in practice use different translations for the same source language feature depending on the context. The motivation is to provide more efficient translations that are sound only in certain cases. This diversity also shows up in the existing Viper-to-Boogie translation. For instance, the existing translation does not emit any Boogie code for the nondeterministic heap assignment as part of an **`exhale`** if the exhaled assertion has no accessibility predicates. This is sound, because in such a case the **`exhale`** does not remove any permissions and thus the nondeterministic heap assignment has no effect.

The existing translation also translates the Viper statement **`assert`** A differently depending on whether A has accessibility predicates. The Boogie code on the far left in Figure 3.10 shows the structure of the existing translation if A has accessibility predicates. First, the translation assigns the current heap variable `H` and permission mask variable `M` to fresh variables `aH` and `aM`, respectively. Then, the translation of **`remcheck`** A follows, where the translation uses `aH` and `aM` as the Boogie variables modelling the Viper heap and permission mask, respectively. In particular, this translation of **`remcheck`** A does not use the original variables `H` and `M`. Finally, the translation of the next statement after the **`assert`** statement is performed again w.r.t. the original variables `H` and `M`. Since the values stored in `H` and `M` are not changed by the translation of **`remcheck`** A, this reflects the fact that **`assert`** A has no effect on the current Viper state.

If A has no accessibility predicates, then the translation does not use fresh variables `aH` and `aM`. Instead, the translation of **`remcheck`** A is done directly with the original variables `H` and `M`. This is sound because in this case **`remcheck`** A does not make changes to the Viper state, and thus `H` and `M` are not affected by the translation.

There are also examples where the existing translation chooses an alternative whose soundness is justified by a condition that is in general hard to check syntactically. In such cases, the translation generates Boogie code with the purpose of checking this condition. For instance, the existing translation omits expression well-definedness checks in the translation of the **`exhale`** of a method call precondition and the **`inhale`** of a method call postcondition, because the translation generates separate Boogie code that checks the *well-formedness* of the pre- and postcondition. As a result, if the pre- and postcondition are not well-formed, then the Viper-to-Boogie implementation reports an error. Well-formedness of a method specification guarantees that expressions evaluated during **`exhale`** and **`inhale`** operations modelling a corresponding method call are always well-defined. Well-formedness is a *semantic condition* that cannot easily

```
method main(x: Ref)
{
  m(x)
}

method m(x: Ref)
  requires mpre
  ensures  mpost
{
  mbody
}
```

```
1  proc bmain(x: bref)
2  {
3    //local var decls elided
4    M := ZeroMask
5    //m(x) start
6    ... /* exhale mpre in H,M
7           (optimised) */
8    ... /* inhale mpost in H,M
9           (optimised) */
10   //m(x) end
11 }
```

```
12 proc bm(x: bref)
13 {
14   //local var decls elided
15   M := ZeroMask
16   ... //inhale mpre in H,M
17   if(*) {
18    pM := ZeroMask
19    havoc pH
20    ... //inhale mpost in pH,pM
21    assume false
22   }
23   ... //mbody in H,M
24   ... /* exhale mpost in H,M
25          (optimised) */
26 }
```

Figure 3.11: The translation of two Viper methods `main` and `m` into two Boogie procedures as emitted by the existing Viper-to-Boogie translation. `bmain` encodes `main` and `bm` encodes `m`. `main` has the trivial pre- and postcondition **`true`**. `m` has the precondition `mpre` and postcondition `mpost`. Three dots in the Boogie procedures indicate an omitted part of the translation for the sake of presentation and the comment next to the dots indicates (1) the Viper construct that is translated in the corresponding dots and (2) which Boogie variables are used in this part of the translation to represent the Viper heap and permission mask. The translation of **`inhale`** and **`exhale`** is optimised when translating a call (lines 6 and 8) and the translation of the **`exhale`** of a postcondition ensuring that the postcondition is satisfied at the end of a method (line 24). The Boogie constant `ZeroMask` represents the permission mask storing zero permissions for all locations; a Boogie axiom not shown here constrains `ZeroMask` accordingly. The translation of a Viper method starts by setting the mask variable `M` representing the Viper permission mask to `ZeroMask` since the correctness of a Viper method is w.r.t. initial states that do not have permissions (Subsection 3.2.6 on page 119).

be checked syntactically in general. We will present the well-formedness definition and discuss why well-formedness justifies the alternative translation of **`exhale`** and **`inhale`** in more detail in Subsection 3.3.4.

The examples discussed so far revolved around alternative translations depending on conditions that may hold. However, there are also alternative translations, which are sound under the same conditions. A formal validation approach should be able to deal with as many of these as possible in case the implementation changes. For instance, when A has accessibility predicates, there are various natural translations for **`assert`** A: the two code examples in the middle and on the far right of Figure 3.10 show natural alternatives to the one used by the existing translation. The code in the middle encodes the **`remcheck`** operation w.r.t. the original variables `H` and `M`, but then resets those variables at the end to the values they had initially via auxiliary variables. The code on the right does not introduce any auxiliary variables at all and instead performs the translation of **`remcheck`** A in a separate branch. Each of these translations would correctly model the semantics of an **`assert`** A statement, and thus, a formal validation approach should ideally be able to justify each of these.

3.3.3. Challenge 3: Non-Locality

So far we have illustrated translations of Viper statements into Boogie statements. For the illustration of the final challenge, we must consider the translation of multiple Viper methods into corresponding Boogie procedures. Figure 3.11 shows the existing Viper-to-Boogie translation of two Viper methods `main` and `m`, each of which is translated to a separate Boogie procedure (`main` is translated to `bmain` and `m` is translated to `bm`).

A key challenge is formally justifying the translation of `main`'s method call `m(x)` in `bmain`. At a high level, the translation reflects the semantics of the call. That is, first the **`exhale`** of `m`'s precondition is translated followed by the translation of the **`inhale`** of `m`'s postcondition. However, in contrast to the usual translation of **`exhale`** and **`inhale`**, the translation for both of these operations as part of a method call *omits* the well-definedness checks for expressions appearing in the pre- and postcondition. For example, for a postcondition `acc(x.f) && y.f > 0`, the well-definedness of `y.f` would not be checked by the translation. This optimisation must be justified, because the Viper semantics specifies that **`exhale`** and **`inhale`** result in failure if an expression evaluated during the operations is not well-defined.

The reason this optimised translation is sound is because the translation of the *callee* checks that the callee's specification is *well-formed*. As we will explain in Subsection 3.3.4, well-formedness of a specification implies that expressions evaluated during the corresponding **`exhale`** and **`inhale`** operations must be well-defined. This is a typical optimisation performed by front-end translations used in practice: a condition is checked once (*e.g.* the well-formedness of a specification) and as a result the translation is optimised at various points (*e.g.* every corresponding method call).

One challenge that such optimisations cause is that to justify the translation of a call as part of one Boogie procedure, one must take into account a check performed in a *different* Boogie procedure. This requires *non-local* reasoning, because one cannot justify the Boogie code within each Boogie procedure locally (*i.e.* independently from all other procedures). Another challenge is setting up a formal validation approach that deals *uniformly* with the standard **`exhale`** (resp. **`inhale`**) translation and the optimised **`exhale`** (resp. **`inhale`**) translation, instead of having two completely separate approaches for the standard and optimised translations. A uniform approach prevents unnecessary duplication and scales to a large variety of possible optimisations and their combinations.

3.3.4. A Closer Look at the Non-Local Method Call Optimisation

In the following, we provide more details regarding the method call optimisation described in Subsection 3.3.3. First, we take a closer look at what it means for a specification to be well-formed and why well-formedness justifies the optimised method call translation at a high level. In later sections, we will make this high-level justification formal. Second, we discuss how the callee's translation checks well-formedness of the callee's method specification.

The following definition states when a specification for a method m is well-formed w.r.t. field declarations F:

Definition 3.3.1 (Well-formedness of a method specifcation)

$$\begin{aligned}
&\mathit{SpecWf}^F(m) \triangleq \\
&\quad \forall \sigma_v.\ [\mathit{stateWellTyVpr}(F, m, \sigma_v) \land \mathit{consistent}(\sigma_v)] \Longrightarrow \\
&\quad\quad \forall r_v.\ \langle \mathit{pre}(m), \sigma_v\rangle \rightarrow_{\mathsf{inh}} r_v \Longrightarrow r_v \neq F \land \\
&\quad\quad\quad \forall \sigma_v'.\ r_v = N(\sigma_v') \Longrightarrow \\
&\quad\quad\quad\quad \forall h, m.\ \text{let } \sigma_v'' = (ST(\sigma_v), h, m) \text{ in} \\
&\quad\quad\quad\quad [\mathit{stateWellTyVpr}(F, m, \sigma_v'') \land \mathit{consistent}(\sigma_v'')] \Longrightarrow \\
&\quad\quad\quad\quad \neg(\langle \mathit{post}(m), \sigma_v''\rangle \rightarrow_{\mathsf{inh}} F)
\end{aligned}$$

Well-formedness of the callee's specification guarantees that for any consistent and well-typed state σ_v (1) inhaling the callee's precondition in σ_v cannot fail ($\langle \mathsf{pre}(m), \sigma_v\rangle \rightarrow_{\mathsf{inh}} r_v \Longrightarrow r_v \neq \mathsf{F}$), and (2) if inhaling the callee's precondition in σ_v leads to a normal outcome ($r_v = \mathsf{N}(\sigma_v')$), then for any consistent and well-typed state σ_v'' that has the same store as σ_v, inhaling the callee's postcondition does not fail in σ_v'' ($\neg(\langle \mathsf{post}(m), \sigma_v''\rangle \rightarrow_{\mathsf{inh}} \mathsf{F})$). Both conditions specify that inhaling an assertion cannot fail. Intuitively, if inhaling an assertion cannot fail *in any state*, then the assertion must provide all permissions required to evaluate expressions in the assertion. In the implicit dynamic frames terminology, this means that the assertion is *self-framing* [33, 101]. So, condition (1) implies that the precondition is self-framing. Condition (2) is a slightly weaker condition, since only those states are considered whose store σ *is compatible* with the precondition (*i.e.* inhaling the precondition leads to a normal outcome from a state whose store is σ). For instance, if inhaling the precondition never leads to a normal outcome (*e.g.* the precondition is `false`), then condition (2) holds trivially. Another example showing the discrepancy between the conditions is the following. Condition (1) does not hold for precondition `acc(x.f) && y.f > 0`, since inhaling this precondition fails in a state where arguments `x` and `y` are different, and where there is no permission to `y.f`. However, condition (2) holds for postcondition `acc(x.f) && y.f > 0` if the precondition is `x == y`, since all states considered in (2) must map `x` and `y` to the same value in this case.

[33]: Smans et al. (2012), *Implicit Dynamic Frames*
[101]: Parkinson et al. (2012), *The Relationship Between Separation Logic and Implicit Dynamic Frames*

Why does well-formedness of the callee's specification justify the optimised method call translation? For the optimised `inhale` translation of the callee's postcondition, this is easier to see, since the property almost directly follows from well-formedness. In the semantics of a call, the `inhale` of a postcondition is relevant only in states in which the `exhale` of the precondition succeeded. As a result, this `inhale` is relevant only in states whose store is compatible with the precondition (we will discuss why in a moment). Therefore, well-formedness guarantees that this `inhale` cannot fail, which means that all expressions evaluated as part of the `inhale` must be well-defined (otherwise the `inhale` would fail). Thus, the optimised `inhale` translation, which omits well-definedness checks, is justified.

To show that the `inhale` of the callee's postcondition is relevant only in states whose store is compatible with the precondition one must show: a successful `exhale` of the precondition implies that the `inhale` of the *precondition* is successful from some state. This result follows from a fundamental connection between `remcheck` (which forms the core of `exhale`) and `inhale`, which we will formally prove in Subsection 3.5.3 on page 159. The connection states that if `remcheck` A succeeds in a state σ_v and we know that `inhale` A cannot fail from a state σ'_v that differs from σ_v only on the permission mask, then inhaling A from σ'_v succeeds and adds precisely those permissions that the `remcheck` A operation removed (as long as adding those permissions does not yield an inconsistent state).

To justify the optimised `exhale` translation of the callee's precondition from well-formedness, one must deal with the fact that well-formedness expresses the condition on the precondition via `inhale` and not `exhale`. To do so, one can use the fundamental connection between `inhale` and `remcheck` described in the previous paragraph. The intuition for the justification is the following. Suppose the `exhale` of the precondition is executed in a Viper state σ_v and suppose an expression e in the precondition is evaluated as part of the corresponding `remcheck` operation. Thus, for the evaluation of e not to fail as part of the `remcheck` operation, σ_v must contain permissions to all heap locations evaluated in e. We know that σ_v must contain all the permissions specified in the precondition *before* e (*i.e.* to the left of e in the precondition) that were removed by the `remcheck` operation, because otherwise the `remcheck` operation would have failed before reaching the evaluation of e. Using the connection between `inhale` and `remcheck`, we get that because the `inhale` of the precondition does not fail (guaranteed by well-formedness), the `inhale` of the precondition from a state with no permissions must also reach the evaluation of e by adding precisely those permissions that the `remcheck` removed before reaching e. Thus, e must be well-defined w.r.t. the added permissions, otherwise the `inhale` of the precondition would fail. Therefore, the optimised translation of the `exhale` is justified, since we know that σ_v contains at least the added permissions.

Checking well-formedness

Let us now take a look at how the callee's translation checks well-formedness of a callee's method specification. The callee's translation includes the specification well-formedness check right at the beginning of the corresponding Boogie procedure, as shown for procedure `bm` in Figure 3.11 on page 127. The translation reflects the definition closely. First, the `inhale` of the precondition is translated on line 16, which captures condition (1) in the well-formedness definition (inhaling the precondition does not fail in any state). Here, the Viper heap and permission mask are captured by Boogie variables `H` and `M`, respectively. Second, the `inhale` of the postcondition is translated in a separate branch on lines 18 - 21, which captures condition (2) in the well-formedness definition (inhaling the postcondition does not fail in any state whose store is compatible with the precondition). Here, the Viper heap and permission mask are captured by fresh Boogie variables `pH` and `pM`, respectively. The Viper variables tracked in the Viper store in this postcondition check are captured by corresponding Boogie variables, which have the same values as during the encoding of condition (1). This reflects the fact that condition (2) considers only Viper states whose store is compatible with the precondition.

There is one discrepancy between the encoding and the well-formedness definition. At the beginning of both `inhale` operations, the Boogie variable tracking the permission mask is initialised to track the empty permission mask, even though the definition does not restrict the Viper states to have no permissions. However, one can prove that if an `inhale` cannot fail from a state without permissions, then it also cannot fail from the same state where the permission mask is changed arbitrarily as long as the inhaled assertion has no permission introspection. Since Viper disallows permission introspection in specifications (except in a specialised advanced construct called *inhale-exhale assertions*, which we do not support), the check thus reflects the definition accurately.

Note that the `inhale` of `m`'s precondition translated on line 16 in procedure `bm` serves two purposes: (1) ensuring the precondition's well-formedness as discussed above, and (2) *additionally* this translation is used to justify the correctness of the method itself. Regarding (2), recall that a Viper method is correct (Definition 3.2.1 on page 119) if inhaling the precondition from every well-typed state without permissions, followed by executing the method body, and finally exhaling the postcondition does not fail. Lines 15-16 encode inhaling the precondition from every well-typed state *without* permissions (this is the motivation for tracking the *empty* permission mask in Boogie via line 15), line 23 encodes executing the method body, and line 24 encodes exhaling the postcondition. Together these lines encode the correctness of the method. The encoded branch for the postcondition's well-formedness does not interfere with the method correctness encoding, since all Boogie executions at the end of the branch are stopped using an `assume false`. Finally, note that the translation of the `exhale` of a postcondition on line 24 to finalise the method correctness check also omits well-definedness checks, which is justified by the well-formedness of the callee's specification.

3.3.5. Extended Boogie Subset

The Boogie subset formalised in Chapter 2 does not fully capture the Boogie subset that the existing Viper-to-Boogie translation targets for the Viper subset formalised in Section 3.2. In particular, the following Boogie features are not formalised in Chapter 2 that are required for the existing Viper-to-Boogie translation: (1) reals to represent permission values, (2) unique constants to represent Viper fields (different unique constants are guaranteed to have different values), and (3) polymorphic maps to represent the Viper state. To deal with this, we additionally formalised reals and unique constants, but did not extend the Boogie certificate generation in Chapter 2 to support these (doing so should be straightforward). To handle the more challenging omission of polymorphic maps, we provide a technique for representing polymorphic maps in our formalised subset, as we will discuss in Subsection 3.3.6.

The formalisation of reals and unique constants is straightforward. For reals, we made the following adjustments to the Isabelle formalisation: (1) added an additional constructor for reals (represented by Isabelle reals) to the algebraic data type definition for Boogie values, (2) added a separate operator for the division between two reals, and (3) added a unary operator that casts integers to reals. The semantics of (2) and (3) is straightforward. Finally, we adjusted the Boogie typing rules to take reals into account, and updated our Boogie type soundness proof and our Isabelle tactic for automatically proving Boogie type judgements.

For unique constants, we needed to adjust our Boogie procedure correctness definition (Definition 2.3.1 on page 32 in Chapter 2) to have another premise stating that values in the initial state corresponding to different unique constants must be different. We made sure that the verification condition produced by Boogie is consistent with this premise by manually inspecting the generated verification condition for examples with unique constants.[13]

13: Note that we needed manual inspection, because we did not extend the Boogie certificate generation for unique constants.

3.3.6. Background Theory and Polymorphic Maps

So far, we have seen examples showing the existing translation of Viper methods into Boogie procedures. In addition to Boogie procedures, the existing translation of a Viper program includes global Boogie declarations (*i.e.* type constructors, functions, global variables, constants, and axioms) in the corresponding Boogie program, which form the Boogie program's background theory.

To justify the existing Viper-to-Boogie translation, one must take these declarations into account. In particular, an important ingredient for this justification is to derive from the correctness of a Boogie procedure p that p's procedure body has no failing executions. Then, using the fact that no procedure has failing executions in a correct Boogie program, one can derive that an input Viper program is correct if the corresponding Boogie program is correct. The correctness of a Boogie procedure p (Definition 2.3.1 on page 32 in Chapter 2) guarantees that p's body cannot fail for *any* interpretation of the uninterpreted types and functions that is well-formed (*e.g.* the function interpretation respects the declared function signatures), and for which all the Boogie axioms in the Boogie program are satisfied in the initial Boogie state (axioms may refer to constants in the state). Thus, to derive that p's body has no failing executions from the correctness of p, we must choose a type interpretation, a function interpretation, and values for each constant such that the choice satisfies the required conditions.

The goal of this subsection is to (1) provide examples for the declarations generated by the existing Viper-to-Boogie translation, and (2) to then discuss our solution to the main challenge for choosing an appropriate type and function interpretation for these declarations, which is dealing with *polymorphic Boogie maps*.

Global Boogie declarations

A subset of the global Boogie declarations always emitted by the existing Viper-to-Boogie translation is given by:

- Uninterpreted types `bref` and `bfield` to model Viper references and fields. `bref` takes no type arguments and `bfield` takes one type argument indicating the type of the corresponding Viper field.[14]

14: In practice, `bfield` takes one more type argument that we ignore for the sake of presentation. This additional type argument is relevant only for features outside of our supported Viper subset.

- An uninterpreted function `GoodMask` that maps a permission map to a Boolean and an axiom restricting this function to return true only if the permission map models a consistent Viper permission mask (see Figure 3.9 on page 124 for an encoding using `GoodMask`).
- An uninterpreted function `idOnPositive`, which maps two heap maps and a permission map to a Boolean, and an axiom restricting this function to return true only if the two heap maps agree on all locations to which the permission map stores positive permission (see Figure 3.9 on page 124 for an encoding using `idOnPositive`).
- Global variables `H` and `M` to model the Viper heap and permission mask, respectively. `H[x,f]` stores the heap value for heap location `x.f` and `M[x,f]` stores the permission value for `x.f` (permission values are represented via Boogie reals). The types of both variables are represented via Boogie's *impredicatively-polymorphic maps* [67], which we explain below.
- Constants to model various Viper constructs such as the **`null`** reference and the empty Viper permission mask. Moreover, there is one Boogie constant per Viper field; these constants modelling fields are marked as unique constants (see Subsection 3.3.5 on page 131) to reflect that different Viper fields are never part of the same heap location.

Figure 3.12 shows a subset of the Boogie background generated by the existing Viper-to-Boogie translation for a Viper program with two integer fields and one Boolean field. The first two axioms constrain the uninterpreted functions `GoodMask` and `idOnPositive`, respectively, and the third axiom constrains the constant `ZeroMask` that models the empty permission mask. In these axioms, `HeapType` and `MaskType` represent the (polymorphic) map types for modelling the Viper heap and the Viper permission mask, respectively.

For the most part, choosing a type interpretation, function interpretation, and constant values to satisfy the required constraints (*e.g.* the axioms hold) for the declarations emitted by the existing Viper-to-Boogie translation is straightforward. For instance, the interpretation of type constructor `bref` can be directly mapped to the Viper reference values. The function `GoodMask` can be interpreted to precisely match the condition imposed by the corresponding axiom: evaluating to true iff the input permission map contains permissions that are most 1. The constant `ZeroMask` can be mapped to the empty permission map.

The main challenge is dealing with the polymorphic map types that model the Viper heap and Viper permission mask (*i.e.* `HeapType` and `MaskType`), since the Boogie formalisation from Chapter 2 does not support maps. Moreover, we are not aware of any existing formalisation of polymorphic maps and it is also not clear whether Isabelle can directly express types that capture every possible polymorphic map. As we will discuss next, we instead encode `HeapType` and `MaskType` into the subset formalised in Chapter 2 by representing these types as Boogie type constructors, and including additional Boogie functions and axioms. As we discuss next, providing a type interpretation for these type constructors is not straightforward.

```
type bref;
type bfield _;

function GoodMask(Mask: MaskType) : bool;
function idOnPositive(H1: HeapType, H2: HeapType, M: MaskType) : bool;

var H: HeapType;
var M: MaskType;

const ZeroMask: MaskType;
const null: ref;

const unique f1: bfield int;
const unique f2: bfield int;
const unique f3: bfield bool;

axiom (
  forall M: MaskType :: GoodMask(M) ==>
    forall x: bref :: forall <T> ::
      forall f: bfield T :: M[x,f] >= 0 && M[x,f] <= 1
);

axiom (
  forall H1: HeapType :: forall H2: HeapType ::
    forall M: MaskType :: idOnPositive(H1,H2,M) ==>
      forall x: bref :: forall <T> :: forall f: bfield T ::
        M[x,f] > 0 ==> H1[x,f] == H2[x,f]
);

axiom (
  forall x: bref :: forall <T> :: forall f: bfield T ::
    ZeroMask[x,f] == 0
);
```

Figure 3.12: A subset of the Boogie background declarations generated by the existing Viper-to-Boogie translation for a Viper program with two integer fields and one Boolean field. `MaskType` and `HeapType` are expressed via polymorphic map types (the concrete polymorphic map types are not shown here). The constants `f1`, `f2`, and `f3` model the three fields. The Boogie axioms shown here constrain the `GoodMask` and `idOnPositive` uninterpreted Boogie functions, and the `ZeroMask` constant. We omit *triggers* for the quantifiers, which the existing translation uses to direct the SMT solver to restrict quantifier instantiations (leveraging SMT solver support for the *E-matching* quantifier instantiation technique [106]). Triggers do not impact our Boogie semantics.

Polymorphic maps

The Viper heap and permission mask are modelled (via the existing Viper-to-Boogie translation) using Boogie's polymorphic maps; this choice is not unusual (*e.g.* a prior version of the existing Dafny-to-Boogie translation also used polymorphic maps with similar polymorphic map types to the ones used by the Viper-to-Boogie translation). The Boogie maps used to model Viper heaps have the polymorphic map type `<T>[bref, bfield T]T`: a total map storing, for *any* type `T`, values of type `T` given (as key) a pair consisting of a reference and a field with type argument `T`.

To our knowledge, there exists no formal model for Boogie's polymorphic maps. Providing a general model is challenging: in particular, Boogie's

```
type HeapType;

function readHeap<T>(h: HeapType, x: bref, f: bfield T) : T;
function updHeap<T>(h: HeapType, x: bref, f: bfield T, v: T) : HeapType;

axiom (
  forall h: HeapType ::
    forall x: bref :: forall <T> :: forall f: bfield T :: forall v: T ::
      readHeap(updHeap(h,x,f,v),x,f) == v
);

axiom (
  forall h: HeapType ::
    forall x: bref :: forall <T> :: forall f: bfield T :: forall v: T ::
      forall y: bref :: forall <T'> :: forall f': bfield T' ::
        (x != y || f != f') ==> readHeap(updHeap(h,x,f,v),y,f') == readHeap(h,y,f')
);
```

Figure 3.13: Global Boogie declarations for modelling the polymorphic map type for the heap.

polymorphic maps are *impredicative*: a map m of type `<T>[T]T'` permits *any* value as a key, including the map m itself! Instead of providing a formal model for polymorphic maps in general, we provide one tailored to those that the existing Viper-to-Boogie translation uses. To aid the incorporation of our model, we adjust the existing translation to represent a polymorphic map via an uninterpreted type (*e.g.* `HeapType` for the heap), polymorphic functions for reading from and updating a polymorphic map (*e.g.* `readHeap` and `updHeap` for the heap), and two axioms that express the relationship between the two functions; Figure 3.13 shows the concrete global declarations emitted for representing the heap. The only change in the translation itself is to simply rewrite heap and mask lookups and updates into calls to these functions; everything else remains identical. The same approach could be used for *e.g.* the prior version of the Dafny-to-Boogie translation.

As discussed above, to derive from the correctness of a Boogie procedure that there are no failing executions of the Boogie procedure body, we need to provide suitable interpretations for these new components (*e.g.* `HeapType`, `readHeap`, and `updHeap` for the heap) such that, in particular, the axioms are fulfilled. The challenge here is avoiding circularities: *e.g.* if the field provided to `readHeap` has type parameter `HeapType`, then the instantiation of `readHeap` must itself return a heap. That is, to construct an initial heap, we already need a heap of the same type. To break this circularity, we instantiate `HeapType` as a *partial* mapping from reference and fields to values, and interpret the empty map to be of type `HeapType`, which provides us with a concrete heap. `readHeap` is defined to return a default value for reference and field pairs that are not in the domain of the partial map (the default value's type matches the type parameter of the field); for the type parameter `HeapType`, the default value is the empty map. This is sufficient to prove the axioms, since the axioms (see Figure 3.13) require `readHeap` returning a specific value for a given heap h and reference-field pair (r, f), only if h is expressible via a sequence of `updHeap` updates and one of those updates inserts a value for (r, f).

Handling polymorphic maps via monomorphisation

An alternative approach to representing the heap in Boogie would be to change the translation to also have an uninterpreted type constructor `HeapType` as in our solution, but to instead add separate *non-polymorphic* functions `readHeapT` and `updHeapT` for *every* concrete type `T` used as a field type in the program. This alternative approach essentially corresponds to a *monomorphisation* of our approach, which desugars the polymorphic function `readHeap` and `updHeap` into multiple non-polymorphic versions. A similar monomorphisation approach of polymorphic maps was added as an option recently by Boogie developers [99]. For the existing Viper-to-Boogie translation, such a monomorphisation solution would circumvent the circular problem of needing to model a heap that itself stores heaps when defining the type interpretation of `HeapType`. The reason is that Viper programs cannot contain fields that themselves store heaps and the existing Viper-to-Boogie translation does not store heaps in fields for encoding purposes. The goal of our solution (with polymorphic update and read functions) is to provide a more-general solution that accurately reflects the polymorphic map type for arbitrary programs (beyond Viper) and different kinds of Boogie encodings (beyond the existing Viper-to-Boogie translation). Thereby, our solution provides additional insights for the formal treatment of polymorphic map types in general.

3.3.7. Instantiating the Abstract Boogie Values and Defining the Type Interpretation

The Boogie values formalised in Chapter 2 are given by the following algebraic data type:

$$'a \ \mathsf{val} \triangleq \mathsf{IntVal}(\mathsf{int}_{isa}) \mid \mathsf{BoolVal}(\mathsf{bool}_{isa}) \mid \mathsf{AbsVal}('a)$$

Here, $\mathsf{AbsVal}('a)$ denotes the abstract Boogie values used to represent inhabitants of the uninterpreted types obtained via the type constructors in a Boogie program. This definition of values is parametric in the carrier type $'a$. In order to choose a type interpretation for the type constructors in a Boogie program, one must first instantiate this carrier type appropriately. In this subsection, we present our instantiation of the carrier type $'a$, and then discuss how to define a suitable type interpretation (*e.g.* the interpretation is well-formed and the axioms are satisfied with such an interpretation) for the Boogie programs generated by the existing Viper-to-Boogie translation.

We instantiate the carrier type using the following algebraic data type absValVpr, which uses an auxiliary algebraic data type absFieldVpr:[15]

15: The data type used in our Isabelle formalisation contains more cases to capture parts of the Boogie encoding that we have not presented here.

$$\begin{aligned}
\mathsf{absFieldVpr} &\triangleq \mathsf{NormalField}(\mathsf{string}_{isa}, \mathit{VType}) \mid \mathsf{OtherField}(\mathit{BType}) \\
\mathsf{absValVpr} &\triangleq \mathsf{ARef}(\mathsf{ref}) \mid \mathsf{AField}(\mathsf{absFieldVpr}) \mid \\
&\quad\ \mathsf{AHeap}(\mathsf{ref} \times \mathsf{absFieldVpr} \rightharpoonup (\mathsf{absValVpr\ val})) \mid \\
&\quad\ \mathsf{AMask}(\mathsf{ref} \times \mathsf{absFieldVpr} \rightarrow \mathsf{real}_{isa}) \mid \\
&\quad\ \mathsf{AOther}(\mathsf{string}_{isa}, \overrightarrow{\mathit{BType}})
\end{aligned}$$

Here, *VType* and *BType* denote the Viper and Boogie types, respectively. Ignore the OtherField$(\cdot)$ and AOther$(\cdot,\cdot)$ constructors for now; we will come back to them later. absValVpr contains separate constructors to represent Viper references, fields, heaps, and permission masks. ref is the algebraic data type for Viper references that is defined in Subsection 3.2.3 on page 108, and so Viper references are represented directly in the instantiation. Fields in absValVpr are represented by the algebraic data type absFieldVpr. A Viper field with identifier f (embedded as a string in Isabelle) and type τ in Viper is represented by the value NormalField(f,τ). Including the field's type τ in this value is advantageous. This type makes clear to which Boogie type NormalField(f,τ) should be interpreted by a type interpretation (namely to the type `bfield`(vprToBoogieTyp(τ)), where `bfield` is the type constructor for the fields discussed in Subsection 3.3.6 and vprToBoogieTyp(τ) is the Boogie type modelling the Viper type τ). Moreover, the field's type also makes clear what the type of the values stored in the Boogie heap representation should be for those fields. The Viper heap is represented as a *partial* map from references and fields to values: as we discussed in Subsection 3.3.6, such a representation deals with the circularity challenge that shows up for a polymorphic Boogie map representing the heap. The range of the partial map is given by the Boogie values where the carrier type is instantiated with absValVpr. Finally, the Viper permission mask is represented as a total map from references and fields to reals. The circularity challenge discussed in Subsection 3.3.6 for the heap does not arise for the permission mask, since the permission mask's range is specified by a primitive type.

A type interpretation for absValVpr is a function from absValVpr to uninterpreted Boogie types (*i.e.* types obtained via type constructors) that are closed (*i.e.* without type variables). There are two points one must take into account when choosing a suitable type interpretation: First, the type interpretation is a *total* (instead of partial) mapping, which means *every* value in absValVpr must be mapped to a type. Changing the semantics to work with a partial type interpretation would lead to additional challenges when formally validating Boogie's verification condition generation (see the end of Subsection 2.6.3 on page 66). Second, the type interpretation must be *well-formed*, which must ensure that *every* possible type is inhabited (see Subsection 2.3.4 on page 31 in Chapter 2). This includes types obtained via type constructors declared in the Boogie program, but which do not model any Viper construct, and types obtained via type constructors that are *not* declared in the Boogie program. Well-formedness of the type interpretation is required to justify Boogie's encoding of functions in the verification condition (see Subsection 2.6.3 on page 66). The interesting cases for choosing a suitable type interpretation deal with both of these points (type interpretation must be total and well-formed). In particular, these cases include the mapping to types for values AHeap(h) representing the heap, and the mapping to types for values obtained from the OtherField$(\cdot)$ and AOther$(\cdot,\cdot)$ constructors, which we have ignored so far. In the following, we discuss our type interpretation for these cases.

We choose to map the value AHeap(h) to a type conditionally depending on properties of the partial function h as follows. If h maps each pair (r,f) in its domain to a Boogie value that matches the type specified by f, then we choose to map AHeap(h) to `HeapType`, which is the uninterpreted type representing Viper heaps. Otherwise, we choose to map AHeap(h) to a closed type τ_{fresh} that does not appear in Boogie programs generated by the existing Viper-to-Boogie translation. With this distinction, we make sure that the underlying partial map of `HeapType` values respects the types of the fields. This allows us, for instance, to show that our *function* interpretation of the Boogie function `readHeap`, which reads from the partial map if the reference-field pair is in the domain, returns a value of the declared return type. The reason we need to map AHeap(h) to a type in the second case (*i.e.* h does not respect the type of at least one field) is because the type interpretation must be total. Note that our type interpretation also uses type τ_{fresh} for the cases discussed below.

Let us now turn our focus to the OtherField($\cdot$) and AOther($\cdot, \cdot$) constructors. The purpose of these constructors is to be able to define a well-formed type interpretation, which must ensure that *every* possible type is inhabited. This includes types obtained via the field type constructor `bfield`, which takes one type argument representing the values stored for that field. If the type argument τ does *not* represent a Viper type, then `bfield`(τ) does not model any Viper field. For instance, `bfield (bfield int)` does not model any Viper field, since `bfield int` does not model a Viper type. To make sure such types are inhabited, we use OtherField(τ). In particular, we choose to map OtherField(τ) to the type `bfield`(τ) if τ is closed and otherwise to τ_{fresh}, which makes sure all types obtained from `bfield` are inhabited. Moreover, we use AOther(C, ts) to ensure every remaining type is inhabited. In particular, if C is a type constructor that does not occur in Boogie programs generated by the existing Viper-to-Boogie translation and ts are closed types, then we choose to map AOther(C, ts) to the type $C(ts)$. Otherwise, we choose to map AOther(C, ts) to τ_{fresh}. This ensures that all types are inhabited.

This concludes our discussion of different aspects of the existing Viper-to-Boogie translation. Next, we will present our simulation methodology that forms the core of our formal validation approach (Section 3.4) and how we apply this methodology to generate certificates for the existing Viper-to-Boogie translation (Section 3.5).

3.4. A Forward Simulation Methodology for Front-End Translations

A front-end translation is *sound* iff the correctness of an input program is implied by the correctness of the correspondingly-translated IVL program. In our setting: a Viper program (resp. a Boogie program) is *correct* if each of its methods (resp. procedures) is correct. At a high level (details in Subsection 3.2.6 on page 119 and Subsection 2.3.4 on page 31), a method (resp. procedure) is correct if its body has no failing executions. Thus, certifying a Viper-to-Boogie translation boils down to proving that *if* the Viper program has a failing execution, then the translated Boogie program has one also.

We generate such certificates *automatically* via a novel general methodology for decomposing and modularly proving *forward simulations* [74] between source and IVL target statements. Our generated certificates

[74]: Lynch et al. (1995), *Forward and Backward Simulations: I. Untimed Systems*

contain sufficient information such that Isabelle can successfully check them *automatically*. We observed early on that generating such certificates directly based on knowledge of the entire translation would require handling the entire semantic gap between the source and target languages monolithically in one result, which would be both infeasible to automate effectively and highly-brittle to any changes in the translation.

Instead, our methodology employs a combination of key strategies that work together to achieve reliable and robust automation of our formal simulation results, and which tackle the challenges for front-end translation outlined in Section 3.1: (1) syntactic and semantic *decompositions* into smaller and more-focused simulation sub-results that are easier to automate, (2) *generic simulation judgements* which can be instantiated to obtain the diverse simulation notions we require, (3) *generic lemmas* involving generic simulation judgements which factor out common idioms arising in diverse facets of the translation, and (4) *contextual hypotheses* which can be injected into simulation proofs to handle the non-locality of certain translation checks. We present these key ingredients of our methodology in this section.

We illustrate these key ingredients for Viper (as the input language) and Boogie (as the IVL), but they can be naturally ported to other input languages and IVLs if one provides a formal semantics for the languages. This is because our ingredients are parametric in the state relations and IVL statements employed in a translation. Another consequence is that the methodology presented in this section allows capturing many translations from Viper to Boogie beyond the existing translation. That is, we do *not* make choices specific to the existing translation in this section; we will consider the application of the methodology to the existing translation in the *next* section (Section 3.5).

Apart from the generality of the methodology w.r.t. different front-end translations, we have also taken care in this section to provide approaches that help with extending the approach to more input language features (*e.g.* to more Viper features than we consider in our application to Viper). We achieve this in three ways, which we will clarify in the section. First, we phrase all of our simulations as instantiations of the same generic simulation judgement, which allows proving results once that can then be composed to obtain results of instantiations without much effort, as opposed to redoing work for each instantiation. Thus, there is a systematic way of saving work when adding support for new features. Second, we decompose simulations into smaller simulations, which allows different Viper features or even different aspects of the same feature to be handled independently from each other. Third, we develop a systematic approach to deal with translations justified by non-local checks, which enables using a single high-level certification strategy for dealing with features that have alternative translations justified by non-local checks. As a result, fewer strategies need to be maintained, which is important when the supported subset increases.

3.4.1. Focusing Forward Simulation Proofs by Decomposition

Intuitively, a forward simulation between a Viper and a Boogie statement shows that for any execution of the Viper statement, there exists a corresponding execution of the Boogie statement that *simulates* it. By defining the simulation such that a *failing* Viper execution is simulated only by *failing* Boogie executions, a forward simulation implies our desired result in particular.

To tackle the complexity of automatically (and reliably) generating simulation proofs in general for the Viper-to-Boogie translation, we employ a variety of strategies for aggressively decomposing the desired simulation result into smaller and simpler sub-goals that are themselves still simulation results. These decompositions are sometimes intuitive based on the syntax: for example, in the case of decomposing the simulation of a Viper sequential composition into simulations for its constituent statements, and in the case of decomposing the simulation of a **remcheck** of a separating conjunction into simulations for the **remcheck** of its two conjuncts. However, we go *further than the syntax*, decomposing across different *semantic concerns* for the *same* Viper statement, into what we call Viper *effects*.

For example, we discussed in Subsection 3.2.5 that the semantics of **exhale** consists of two effects, **remcheck** and a nondeterministic assignment. The simulation proofs for each of these Viper effects are made separately, and then composed for a simulation proof for the primitive statement as a whole; this would in turn be composed with simulation proofs for other sequentially-composed statements, and so on. The simulation proof for the **remcheck** effect is further split, for instance, into simulation proofs for different conjuncts. Note in particular, that simulation proofs may need to relate only a *part of* the semantics of a Viper statement to some appropriate Boogie code. This requires adding sufficient contextual information to such simulations, which captures how the simulation relates to the simulation of the entire statement.

Via our decompositions, each resulting simulation proof focuses on a different specific semantic concern with respect to the translation in question; these proofs can be made simple enough to discharge automatically, optionally with tailored tactics. Apart from enabling automation, another advantage of our decompositions is that it reduces the effort significantly when adapting the simulation proof to local changes in the translation and extending simulation proofs to more input language constructs. Adapting the simulation proof to local changes in the translation in many cases affects only the part that is changed, since our decomposition separates different semantic concerns into different simulations. Similarly, adding support for the translation of a new input language construct allows one to focus on that language construct in isolation (if the construct does not fundamentally change how to reason about the remaining constructs).

Without care, our decomposition approach could lead easily to an explosion of ad hoc simulation judgements with disparate forms and parameters. Instead, our simulation methodology defines a *single, generic* simulation judgement which can be instantiated appropriately to define

$$
\begin{aligned}
&\mathsf{sim}_{\Gamma_b}(R_{in}, R_{out}, \mathit{Succ}, \mathit{Fail}, \gamma_{in}, \gamma_{out}) \triangleq \forall \tau, \sigma_b.\ R_{in}(\tau, \sigma_b) \Longrightarrow \\
&\quad \big(\forall \tau'.\ \mathit{Succ}(\tau, \tau') \Longrightarrow \exists \sigma_b'.\ \Gamma_b \vdash (\gamma_{in}, \mathsf{N}(\sigma_b)) \rightarrow^*_{\mathsf{AST2}} (\gamma_{out}, \mathsf{N}(\sigma_b')) \wedge R_{out}(\tau', \sigma_b')\big) \wedge && \text{(Success case)} \\
&\quad \big(\mathit{Fail}(\tau) \Longrightarrow \exists \gamma'.\ \Gamma_b \vdash (\gamma_{in}, \sigma_b) \rightarrow^*_{\mathsf{AST2}} (\gamma', \mathsf{F})\big) && \text{(Failure case)}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{stmSim}_{\Gamma_v,\Gamma_b}(R, R', s, \gamma, \gamma') \triangleq \\
&\quad \mathsf{sim}_{\Gamma_b}(R, R', \lambda \sigma_v\ \sigma_v'.\ \Gamma_v \vdash \langle s, \sigma_v \rangle \rightarrow_{\mathsf{v}} \mathsf{N}(\sigma_v'), \lambda \sigma_v.\ \Gamma_v \vdash \langle s, \sigma_v \rangle \rightarrow_{\mathsf{v}} \mathsf{F}, \gamma, \gamma')
\end{aligned}
$$

$$
\mathsf{wfSim}_{\Gamma_b}(R, R', e, \gamma, \gamma') \triangleq \mathsf{sim}_{\Gamma_b}\left(\begin{array}{l} R, R', \\ (\lambda(\sigma_v^0, \sigma_v)\ (\sigma_v^1, \sigma_v').\ (\sigma_v^0, \sigma_v) = (\sigma_v^1, \sigma_v') \wedge \exists v.\ \sigma_v^0 \vdash \langle e, \sigma_v \rangle \Downarrow_{\mathsf{v}} \mathsf{V}(v)), \\ (\lambda(\sigma_v^0, \sigma_v).\ \sigma_v^0 \vdash \langle e, \sigma_v \rangle \Downarrow_{\mathsf{v}} \lightning), \gamma, \gamma' \end{array}\right)
$$

$$
\mathsf{wfSimList}_{\Gamma_b}(R, R', es, \gamma, \gamma') \triangleq \mathsf{sim}_{\Gamma_b}\left(\begin{array}{l} R, R', \\ \left(\begin{array}{l} \lambda(\sigma_v^0, \sigma_v)\ (\sigma_v^1, \sigma_v').\ (\sigma_v^0, \sigma_v) = (\sigma_v^1, \sigma_v') \wedge \\ \exists vs.\ \sigma_v^0 \vdash \langle es, \sigma_v \rangle\ [\Downarrow]_{\mathsf{v}}\ \mathsf{V}(vs) \end{array}\right), \\ (\lambda(\sigma_v^0, \sigma_v).\ \sigma_v^0 \vdash \langle es, \sigma_v \rangle\ [\Downarrow]_{\mathsf{v}} \lightning), \gamma, \gamma' \end{array}\right)
$$

$$
\mathsf{inhSim}_{\Gamma_b}(R, R', A, \gamma, \gamma') \triangleq \mathsf{sim}_{\Gamma_b}(R, R', (\lambda \sigma_v\ \sigma_v'.\ \langle A, \sigma_v \rangle \rightarrow_{\mathsf{inh}} \mathsf{N}(\sigma_v')), (\lambda \sigma_v.\ \langle A, \sigma_v \rangle \rightarrow_{\mathsf{inh}} \mathsf{F}), \gamma, \gamma')
$$

$$
\mathsf{rcSim}_{\Gamma_b}(R, R', A, \gamma, \gamma') \triangleq \mathsf{sim}_{\Gamma_b}\left(\begin{array}{l} R, R', (\lambda(\sigma_v^0, \sigma_v)\ (\sigma_v^1, \sigma_v').\ \sigma_v^0 = \sigma_v^1 \wedge \sigma_v^0 \vdash \langle A, \sigma_v \rangle \rightarrow_{\mathsf{rc}} \mathsf{N}(\sigma_v')), \\ (\lambda(\sigma_v^0, \sigma_v).\ \sigma_v^0 \vdash \langle A, \sigma_v \rangle \rightarrow_{\mathsf{rc}} \mathsf{F}), \gamma, \gamma' \end{array}\right)
$$

Figure 3.14: The definition of the generic forward simulation judgement sim and five common instantiations. Note that the Boogie AST reduction is expressed using the reflexive-transitive closure of the $\rightarrow_{\mathsf{AST2}}$ one-step reduction, which reduces each basic command in a separate step instead of reducing the list of basic commands at the beginning of a statement block in one step (see Subsection 2.8.2 on page 81 in Chapter 2), providing more flexibility for simulations.

each particular simulation judgement required. This allows us to prove lemmas that capture different idioms (*e.g.* composition, conditional evaluation, *etc.*) once using our generic simulation judgement. These generic lemmas can then be reused for different instantiations; Subsection 3.4.3 presents some examples of generic lemmas. We design our generic judgements to support instantiations which reflect not only the semantics of the particular effect in isolation, but to optionally include additional contextual information to specialise and aid the simulation proof itself.

3.4.2. One Simulation Judgement to Rule Them All

Our generic forward simulation judgement sim is defined in Figure 3.14. All concrete forward simulations (*e.g.* for statements, well-definedness checks, etc.) are instantiations of this judgement. As well as aiding understanding, this approach enables both tactics which manipulate this generic judgement directly, and *generic composition proof rules* which embody recurring proof idioms that are applicable to different concrete forward simulations (Subsection 3.4.3).

The simulation judgement sim is defined in terms of multiple parameters: (1) a Boogie context Γ_b, (2) an *input relation* R_{in} and an *output relation* R_{out} on Viper and Boogie states, (3) a *success predicate Succ* characterising the set of input and output Viper state pairs (τ, τ') for which there is a successful Viper execution from τ to τ', (4) a *failure predicate Fail* characterising the set of input Viper states that result in a failing execution, (5) input and output Boogie program points γ_{in} and γ_{out} where the Boogie executions are expected to start and end, respectively. The success and failure predicate together abstractly describe the set of Viper executions that must be shown to be simulated.

$\mathsf{sim}_{\Gamma_b}(R_{in}, R_{out}, Succ, Fail, \gamma_{in}, \gamma_{out})$ holds iff for any Viper and Boogie input states related by R_{in}, the following two conditions hold: (1) for any successful Viper execution from the input Viper state to an output Viper state τ', there must be a Boogie execution from program point γ_{in} and the input Boogie state to program point γ_{out} and some output Boogie state that is related to τ' by R_{out}, and (2) if the Viper execution fails starting from the input state, then there must be a failing Boogie execution from γ_{in} and the input Boogie state (the reached Boogie program point need not be γ_{out}). The second condition is the end goal that we need, to show soundness of the Viper-to-Boogie translation. The first condition is needed in order to derive sim compositionally; it guarantees, for example, that not all Boogie executions for a successful Viper execution produce a magic outcome.

sim abstracts over the concrete representation of Viper states: any instantiation for the representation of a Viper state works. We require this abstraction, since our instantiations of sim use two different instantiations for the representation of a Viper state: (1) the instantiation given by the standard notion of a Viper state introduced in Subsection 3.2.3 on page 108, and (2) an instantiation using a pair of the standard Viper states since both the expression evaluation judgement and the **`remcheck`** reduction are defined via two standard Viper states. Abstracting over the representation of Viper states in sim is possible, since the success and failure predicates are parameters of sim.

Five key instantiations of sim that we use heavily are shown at the bottom of Figure 3.14. The instantiation stmSim is the forward simulation for Viper statements, where the representation of the Viper state in sim is instantiated to be the standard notion of a Viper state introduced in Subsection 3.2.3 on page 108. The success predicate is instantiated to hold iff there is a successful Viper statement reduction from the input to the output state, and the failure predicate is instantiated to hold iff there is a failing Viper statement reduction from the input state. Thus, the resulting failure case in sim directly gives us the key property to show the soundness of a Viper-to-Boogie translation. The instantiation inhSim is the forward simulation for the reduction of an **`inhale`** operation, where the representation of the Viper state in sim is instantiated to be the standard notion of a Viper state. wfSim is the forward simulation for the well-definedness check of a Viper expression. The instantiation wfSimList is the analogous simulation for a list of Viper expressions. In both of these instantiations, the representation of the Viper state in sim is instantiated to be a pair of standard Viper states, where the first state represents the permission definedness state and the second state represents the evaluation state (see Subsection 3.2.4 for this distinction). The instantiation of the success predicate explicitly expresses that neither Viper state changes during the evaluation of expressions. The instantiation rcSim is the forward simulation for **`remcheck`** and instantiates the representation of the Viper state in sim via a pair of standard Viper states as wfSim does. The first state represents the permission definedness state and the second state represents the reduction state (see Subsection 3.2.5 for this distinction). The success predicate expresses that the permission definedness state does not change during a **`remcheck`** operation.

$$\dfrac{\begin{array}{c}\mathsf{sim}_{\Gamma_b}(R, R', S_1, F_1, \gamma, \gamma') \\ \mathsf{sim}_{\Gamma_b}(R', R'', S_2, F_2, \gamma', \gamma'') \\ \forall \tau, \tau''.\ S(\tau, \tau'') \Rightarrow \exists \tau'.\ S_1(\tau, \tau') \wedge S_2(\tau', \tau'') \\ \forall \tau.\ F(\tau) \Rightarrow F_1(\tau) \vee \exists \tau'.\ S_1(\tau, \tau') \wedge F_2(\tau')\end{array}}{\mathsf{sim}_{\Gamma_b}(R, R'', S, F, \gamma, \gamma'')}\ (\textsc{comp}) \qquad \dfrac{\begin{array}{c}\mathsf{bSim}_{\Gamma_b}(R, R_1, \gamma, \gamma_1) \\ \mathsf{sim}_{\Gamma_b}(R_1, R_2, S, F, \gamma_1, \gamma_2) \\ \mathsf{bSim}_{\Gamma_b}(R_2, R', \gamma_2, \gamma')\end{array}}{\mathsf{sim}_{\Gamma_b}(R, R', S, F, \gamma, \gamma')}\ (\textsc{bprop})$$

$$\dfrac{\begin{array}{c}\mathsf{stmSim}_{\Gamma_v, \Gamma_b}(R, R', s_1, \gamma, \gamma') \\ \mathsf{stmSim}_{\Gamma_v, \Gamma_b}(R', R'', s_2, \gamma', \gamma'')\end{array}}{\mathsf{stmSim}_{\Gamma_v, \Gamma_b}(R, R'', (s_1; s_2), \gamma, \gamma'')}\ (\textsc{seq-sim}) \qquad \textit{where} \qquad \begin{array}{l}\mathsf{bSim}_{\Gamma_b}(R, R', \gamma, \gamma') \triangleq \\ \mathsf{sim}_{\Gamma_b}(R, R', \lambda \tau\ \tau'.\ \tau = \tau', \lambda\_.\ \bot, \gamma, \gamma')\end{array}$$

$$\dfrac{\begin{array}{c}\mathsf{sim}_{\Gamma_b}(R'_0, R'_1, S', F', \gamma, \gamma') \\ \forall \sigma_v\ \sigma_b.\ R_0(\sigma_v, \sigma_b) \Rightarrow R'_0(\sigma_v, \sigma_b) \quad \text{(weaker input relation)} \\ \forall \sigma_v\ \sigma'_v\ \sigma_b.\ R_0(\sigma_v, \sigma_b) \Rightarrow S(\sigma_v, \sigma'_v) \Rightarrow S'(\sigma_v, \sigma'_v) \quad \text{(weaker success predicate)} \\ \forall \sigma_v\ \sigma_b.\ R_0(\sigma_v, \sigma_b) \Rightarrow F(\sigma_v) \Rightarrow F'(\sigma_v) \quad \text{(weaker failure predicate)} \\ \forall \sigma_v\ \sigma'_v\ \sigma_b\ \sigma'_b.\ R_0(\sigma_v, \sigma_b) \Rightarrow S(\sigma_v, \sigma'_v) \Rightarrow R'_1(\sigma'_v, \sigma'_b) \Rightarrow R''_1(\sigma'_v, \sigma'_b) \quad \text{(stronger output relation)}\end{array}}{\mathsf{sim}_{\Gamma_b}(R_0, R''_1, S, F, \gamma, \gamma')}\ (\textsc{conseq})$$

Figure 3.15: The instantiation-independent rules COMP, BPROP, and CONSEQ, and the concrete rule SEQ-SIM for the simulation of $s_1; s_2$.

These five common instantiations are all expressed directly via the Viper reduction judgements introduced in Section 3.2. Like the generic simulation judgement, these five instantiations *are themselves generic*, abstracting away *how* the Viper and Boogie states are related by taking the input and output state relations as parameters. As we will show in Subsection 3.4.4, we also use instantiations of the success and failure predicates that do not just use Viper reduction judgements (*e.g.* to express the nondeterministic assignment of heap values in `remcheck`).

3.4.3. Instantiation-Independent Rules

Many simulation idioms arise repeatedly in a certification proof for a complex translation. Notions of sequential composition, conditional evaluation and stuttering steps are all good examples, which require a certain stylised formal justification to reason about. Our generic simulation judgement allows us to identify and formalise these idioms once and for all, providing, for example, generic composition lemmas that can be proved once and instantiated for different purposes. As a result, there is less effort involved than when one needs to formalise the idioms from scratch for every instantiation, thus making it easier to maintain larger language subsets. In this subsection, we present these idioms as inference rules, but in our formalisation they are expressed and proved as regular lemmas.

For example, we prove a single general composition rule from which we derive concrete rules to combine (1) simulations of s_1 and s_2 to a simulation of $s_1; s_2$, (2) simulations of `remcheck` A_1 and `remcheck` A_2 to `remcheck` A_1 && A_2, (3) simulations of `inhale` A_1 and `inhale` A_2 to `inhale` A_1 && A_2. The general composition rule COMP in Figure 3.15 captures the composition of two different instantiations of sim, where the output relation and Boogie program point of the first instantiation match the input relation and program point of the second one. The two final premises constrain the resulting success and failure predicates. In particular, the composed Viper execution should fail only if either the first instantiation fails or if the second instantiation fails in a state successfully reached by the first one. The rule SEQ-SIM in Figure 3.15 shows the concrete composition rule for $s_1; s_2$, which is derived from COMP. Note that SEQ-SIM does not impose any constraints on the Boogie program points, which is crucial to handle Viper's and Boogie's disparate ASTs (see Subsection 3.2.1). We will discuss in Subsection 3.5.4 how we deal with the AST mismatch when automating proofs.

As another example, the notion of simulation *stuttering steps* also arises in many ways, whenever some Boogie code is generated that does not fully correspond to a step in the Viper source. This includes initialisations of auxiliary variables, or Boogie `assume` commands for properties implied by the current simulation state relation. This idiom is captured by the *Boogie propagation rule* BPROP in Figure 3.15, in which bSim expresses simulations in which the Viper state remains unchanged, and thus only the Boogie state may change (potentially causing adjustment to the state relations). The rule permits simulating Viper executions by first taking steps only in the Boogie code, then simulating the Viper executions, and finally taking steps only in the Boogie code.

As a final example, the rule CONSEQ in Figure 3.15 shows a consequence rule for the generic forward simulation judgement. The rule enables proving a simulation judgement by proving a different simulation judgement where the input state relation, success predicate, and failure predicate are weakened, and the output state relation is strengthened. For the weakening and strengthening conditions, one may assume that the initial states are related w.r.t. the original input state relation, since all guarantees of the simulation judgement are w.r.t. such initial states. For the strengthening of the output state relation, one may additionally assume that the Viper execution is successful, since the output state relation is relevant only in case of success.

3.4.4. Examples: Generic Decomposition in Action

As outlined above, the general strategy for our simulation methodology is to decompose our simulation goals as far as possible, while leaving as many parameters generic as we can to enable maximal reuse of our results. While decomposition handles the semantic gap, our use of generic parameterisation provides the abstraction to address the diverse translations used in practical translational verifiers. In the following, we showcase our methodology on a selection of rules, but the same ideas apply to all our formal rules.[16] In particular, note that these rules are not specific to the existing Viper-to-Boogie translation, and can be applied to a large variety of different translations.

16: These rules are lemmas proved in our Isabelle formalisation.

Consider the rule EXH-SIM for the simulation of `exhale` A in Figure 3.16. This rule decomposes the simulation of `exhale` A into two Viper effects, each of which is modelled in a premise. The first premise models the simulation of the first effect, `remcheck`, which we can express via the rcSim instantiation (see Figure 3.14 on page 141). The second premise models the simulation of the second effect, the nondeterministic heap

$$\dfrac{\begin{array}{ll}\mathsf{rcSim}_{\Gamma_b}([\lambda(\sigma_v^0, \sigma_v)\ \sigma_b.\ \sigma_v^0 = \sigma_v \wedge R(\sigma_v, \sigma_b)], R', A, \gamma, \gamma') & (\text{sim. of } \texttt{remcheck}\ A) \\ \mathsf{sim}_{\Gamma_b}(R', [\lambda(\_, \sigma_v)\ \sigma_b.\ R''(\sigma_v, \sigma_b)], \mathit{Succ}_2, \lambda\_.\ \bot, \gamma', \gamma'') & (\text{non-det. selection})\end{array}}{\mathsf{stmSim}_{\Gamma_v,\Gamma_b}(R, R'', \texttt{exhale}\ A, \gamma, \gamma'')}\ (\textsc{exh-sim})$$

$$\mathit{Succ}_2 \triangleq \lambda(\sigma_v^0, \sigma_v)\ (\_, \sigma_v').\ \mathsf{nonDet}(\sigma_v^0, \sigma_v, \sigma_v') \wedge \sigma_v^0 \vdash \langle A, \sigma_v^0 \rangle \rightarrow_{\mathsf{rc}} \mathsf{N}(\sigma_v)$$

Figure 3.16: Rule for the simulation of `exhale` *A*. The definition of nonDet is given in Figure 3.6 on page 114.

assignment, which is captured by the first conjunct nonDet of the corresponding success predicate and by the failure predicate, which reflects that the nondeterministic assignment cannot fail. The two simulations are connected by the fact that the output state relation R' and output program point γ' of the first premise match the input relation and program point of the second premise.

By modularly abstracting over the details of these premises, and the precise definitions of the states and state relations (*e.g.* the intermediate relation R' in this rule), we obtain robustness to diverse translations: our rules do not constrain *which exact Boogie statements* correspond to a Viper effect. For example, the Viper-to-Boogie implementation establishes the nondeterministic heap assignment premise in EXH-SIM in two different ways depending on whether the assertion contains an accessibility predicate `acc`$(e.f, p)$ or not; if not, then the implementation does not emit any Boogie code for the nondeterministic assignment, which is sound, since no permission is removed.

Note that this genericity does not prevent the rule from exploiting contextual information. For example, the input state relation of the first premise specifies that at the beginning of the `remcheck` *A* effect the permission definedness state and the reduction state are the same. This property does not hold in general for executions of `remcheck` (*e.g.* it might not hold when executing the second conjunct of a separating conjunction), but it does hold here, at the beginning of an `exhale`. Without this property, it would not be possible to prove the first premise in general for a concrete Boogie encoding, because there would be no information about the permission definedness state.

The second premise's success predicate also includes contextual information. It includes the fact that the current Viper state can be reached via `remcheck` *A*. (This fact is expressed by using that the permission definedness state in Succ_2 is the same as the initial state in which `remcheck` *A* is reduced, since the permission definedness state does not change during the reduction.) This fact allows us, for example, to conclude that the nondeterministic assignment has no effect if `remcheck` *A* removes no permissions, which is required to justify the case when the implementation does not emit Boogie code for the nondeterministic assignment.

The rule ASSERT-SIM for the simulation of `assert` *A* shown in Figure 3.17 is similar to the rule EXH-SIM for `exhale` *A* in Figure 3.16. ASSERT-SIM also decomposes the simulation into two effects. The simulation of the first effect shown in the first premise models the simulation of the `remcheck` effect and is identical to the first premise in EXH-SIM. However, the simulation of the second effect shown in the second premise of ASSERT-SIM is different from the second premise in EXH-SIM in interesting ways.

$$\dfrac{\begin{array}{ll} \mathsf{rcSim}_{\Gamma_b}([\lambda(\sigma_v^0, \sigma_v)\ \sigma_b.\ \sigma_v^0 = \sigma_v \wedge R(\sigma_v, \sigma_b)], R', A, \gamma, \gamma') & \text{(sim. of } \texttt{remcheck}\ A\text{)} \\ \left(\begin{array}{l} \mathsf{sim}_{\Gamma_b}(R', [\lambda(\_, \sigma_v)\ \sigma_b.\ R''(\sigma_v, \sigma_b)], \mathit{Succ}_2, \lambda\_.\ \bot, \gamma', \gamma'') \vee \\ \mathsf{bSim}_{\Gamma_b}(R, R'', \gamma, \gamma'') \end{array}\right) & \left(\begin{array}{l} \text{continue with} \\ \text{previous Viper state} \end{array}\right) \end{array}}{\mathsf{stmSim}_{\Gamma_v,\Gamma_b}(R, R'', \texttt{assert}\ A, \gamma, \gamma'')}\ (\textsc{assert-sim})$$

$$\mathit{Succ}_2 \triangleq \lambda(\sigma_v^0, \sigma_v)\ (\_, \sigma_v').\ \sigma_v' = \sigma_v^0 \wedge \sigma_v^0 \vdash \langle A, \sigma_v^0 \rangle \rightarrow_{\mathsf{rc}} \mathsf{N}(\sigma_v)$$

Figure 3.17: Rule for the simulation of `assert` A. bSim is defined in Figure 3.15.

In the second premise of ASSERT-SIM, the first disjunct models a Viper effect which resets the Viper state reached *after* a successful `remcheck` A operation (modelled in the first premise) to the state *before* the `remcheck` A operation was executed. This effect is captured in the success predicate Succ_2 by using the same observation as in rule EXH-SIM: the permission definedness state at the end of the `remcheck` operation is the same as the state *before* the `remcheck` operation. This state resetting effect models the semantics of a successful `assert` A operation, whose resulting state is given by the initial state. The second premise's second disjunct states that alternatively the Boogie code could simulate Viper executions *after* `assert` A (*i.e.* from the output Boogie program point γ'' onwards) by taking steps in the Boogie program from the *initial* program point γ to the output program point γ'' *while still being related to the Viper state before the* `assert` A operation. This second disjunct is expressed via the bSim simulation instantiation, where the Viper state remains unchanged. Intuitively, the second disjunct expresses that the Boogie code need not simulate resetting the Viper state after successfully simulating `remcheck` A via some execution, if there is a potentially different execution that reaches the output Boogie program point γ'' and that captures the same Viper state as before the `assert` operation.

ASSERT-SIM is abstract enough to capture all three translations of `assert` that we discussed in Subsection 3.3.2 (Figure 3.10 on page 126 illustrates the translations), which illustrates the diversity of translations that can be captured by our technique. For the first two translations (on the left and in the middle in Figure 3.10), the second premise's first disjunct justifies resetting the Viper state after the `remcheck` operation. In the first translation, the state reset is implicit in the Boogie code (*i.e.* there is no Boogie command for resetting the state), which means no steps need to be taken in the Boogie code, but the state relation still changes from R' to R to reflect that, at the end, the Boogie variables `H` and `M` capture the Viper heap and Viper permission mask, respectively. In the second translation, the state reset is done explicitly via assignments. For the third translation (on the right of Figure 3.10), the first disjunct is not applicable, because every Boogie execution that simulates the `remcheck` operation stops right after due to the `assume false` command. Instead, the second premise is able to justify the the third translation: Boogie executions that do not go into the then-branch simulate Viper executions after the `assert` statement.

Finally, note that the success predicate Succ_2 in ASSERT-SIM contains the same contextual information as in the rule for `exhale`: that is, the reduction state σ_v can be reached by reducing `remcheck` A from the permission definedness state. This information is required to justify the

$$\frac{\begin{array}{c}\mathsf{wfSim}_{\Gamma_b}(R, R, e, \gamma, \gamma') \quad \text{(receiver well-defined)} \\ \mathsf{sim}_{\Gamma_b}(R, R, \mathit{Succ}, \mathit{Fail}, \gamma', \gamma'') \quad \text{(sufficient permission to read)}\end{array}}{\mathsf{wfSim}_{\Gamma_b}(R, R, e.f, \gamma, \gamma'')}\ (\textsc{field-wf-sim})$$

$$\mathit{Succ} \triangleq \lambda(\sigma_v^0, \sigma_v)(\sigma_v^1, \sigma_v').\ (\sigma_v^0, \sigma_v) = (\sigma_v^1, \sigma_v') \wedge \exists a.\ \sigma_v^0 \vdash \langle e, \sigma_v \rangle \Downarrow_{\mathsf{v}} \mathsf{V}(\mathsf{VRefVal}(\mathsf{Address}(a))) \wedge \Pi(\sigma_v^0)(a, f) > 0$$
$$\mathit{Fail} \triangleq \lambda(\sigma_v^0, \sigma_v).\ \exists r.\ \sigma_v^0 \vdash \langle e, \sigma_v \rangle \Downarrow_{\mathsf{v}} \mathsf{V}(\mathsf{VRefVal}(r)) \wedge (r = \mathsf{Null} \vee \exists a.\ r = \mathsf{Address}(a) \wedge \Pi(\sigma_v^0)(a, f) = 0)$$

$$\frac{\begin{array}{c}\mathsf{wfSimList}_{\Gamma_b}(\hat{R}, \hat{R}, [e, e'], \gamma, \gamma_1) \quad \text{(receiver and RHS well-defined)} \\ \mathsf{sim}_{\Gamma_b}(\hat{R}, \hat{R}, \mathit{Succ}_2, \mathit{Fail}_2, \gamma_1, \gamma_2) \quad \text{(sufficient permission to write)} \\ \mathsf{sim}_{\Gamma_b}(R, R, \mathit{Succ}_3, \lambda\_.\ \bot, \gamma_2, \gamma') \quad \text{(heap update)}\end{array}}{\mathsf{stmSim}_{\Gamma_v, \Gamma_b}(R, R, e.f := e', \gamma, \gamma')}\ (\textsc{field-assign-sim})$$

$$\hat{R} = \lambda(\sigma_v^0, \sigma_v)\ \sigma_b.\ \sigma_v^0 = \sigma_v \wedge R(\sigma_v, \sigma_b)$$
$$\mathit{Succ}_2 \triangleq \lambda(\sigma_v^0, \sigma_v)(\sigma_v^1, \sigma_v').\ (\sigma_v^0, \sigma_v) = (\sigma_v^1, \sigma_v') \wedge \exists a.\ \sigma_v^0 \vdash \langle e, \sigma_v \rangle \Downarrow_{\mathsf{v}} \mathsf{V}(\mathsf{VRefVal}(\mathsf{Address}(a))) \wedge \Pi(\sigma_v^0)(a, f) = 1$$
$$\mathit{Fail}_2 \triangleq \lambda(\sigma_v^0, \sigma_v).\ \exists r.\ \sigma_v^0 \vdash \langle e, \sigma_v \rangle \Downarrow_{\mathsf{v}} \mathsf{V}(\mathsf{VRefVal}(r)) \wedge (r = \mathsf{Null} \vee \exists a.\ r = \mathsf{Address}(a) \wedge \Pi(\sigma_v^0)(a, f) < 1)$$
$$\mathit{Succ}_3 \triangleq \lambda\sigma_v\ \sigma_v'.\ \exists a\ v. \left(\begin{array}{l}\sigma_v' = (\mathsf{ST}(\sigma_v), \mathsf{H}(\sigma_v)((a, f) \mapsto v), \Pi(\sigma_v)) \wedge \\ \sigma_v \vdash \langle e, \sigma_v \rangle \Downarrow_{\mathsf{v}} \mathsf{V}(\mathsf{VRefVal}(\mathsf{Address}(a))) \wedge \sigma_v \vdash \langle e', \sigma_v \rangle \Downarrow_{\mathsf{v}} \mathsf{V}(v) \wedge \\ \Pi(\sigma_v)(a, f) = 1 \wedge \mathsf{Fields}(\Gamma_v)(f) = \mathsf{typVpr}(v)\end{array}\right)$$

Figure 3.18: Simulation rules for the well-definedness of a field access and for the execution of a field assignment. Note that here the input and output state relations in the conclusion are the same; one could generalise the rules to allow them to be different.

existing translation of `assert` A in the case when A has no accessibility predicates, which is different from the case when A has accessibility predicates.

Both rules presented so far showcase our decomposition into smaller simulations and the genericity of the rules in terms of the state relation and the simulating Boogie code (both of which are parameters). These properties are a recurrent theme in our rules. For two more examples, consider the simulation rules shown in Figure 3.18: rule field-wf-sim handles the well-definedness of a field access and field-assign-sim handles a field assignment. Both rules decompose the simulation similarly: the well-definedness of subexpressions and the check for sufficient permissions are handled in separate simulations. The rule for the field assignment additionally models the simulation of a Viper heap update. The check for sufficient permissions and the Viper heap update simulation are two further examples for custom instantiations that are not defined directly via Viper's reduction judgements. Moreover, both rules use contextual information in the simulations specified in the premises of the rules, such as the receiver being well-defined. Finally, as in our previous rules, these rules abstract over how Boogie encodes the separate effects, which makes the rules applicable to diverse translations.

$$\frac{\begin{array}{c}\mathsf{wfSimList}_{\Gamma_b}(R, R, [e, e_p], \gamma, \gamma_1) \quad \text{(subexpression well-definedness)} \\ \forall r, p.\ \mathsf{sim}_{\Gamma_b}(R, R_B(r,p), \mathit{Succ}_A(r,p), \mathit{Fail}_A(r,p), \gamma_1, \gamma_2) \quad \text{(non-failure check)} \\ \forall r, p.\ \mathsf{sim}_{\Gamma_b}(R_B(r,p), R', \mathit{Succ}_B(r,p), (\lambda\_.\ \bot), \gamma_2, \gamma') \quad \text{(state update)}\end{array}}{\mathsf{rcSim}_{\Gamma_b}(R, R', \mathtt{acc}(e.f, e_p), \gamma, \gamma')} \ (\textsc{racc-sim})$$

$$\mathit{Succ}_A(r,p) \triangleq \left(\lambda(\sigma_v^0, \sigma_v)\ (\sigma_v^1, \sigma_v').\ \begin{array}{l}\mathsf{exhAccSucc}(r, p, \sigma_v) \land (\sigma_v^0, \sigma_v) = (\sigma_v^1, \sigma_v') \land \\ \mathsf{wfAccSucc}(e, e_p, r, p, \sigma_v^0)\end{array}\right)$$

$$\mathit{Fail}_A(r,p) \triangleq \lambda(\sigma_v^0, \sigma_v).\ \neg\mathsf{exhAccSucc}(r, p, \sigma_v) \land \mathsf{wfAccSucc}(e, e_p, r, p, \sigma_v^0)$$

$$\mathit{Succ}_B(r,p) \triangleq \left(\lambda(\sigma_v^0, \sigma_v)\ (\sigma_v^1, \sigma_v').\ \begin{array}{l}\sigma_v' = \mathsf{rem}(\sigma_v, r, f, p) \land \sigma_v^0 = \sigma_v^1 \land \\ \mathsf{exhAccSucc}(r, p, \sigma_v) \land \mathsf{wfAccSucc}(e, e_p, r, p, \sigma_v)\end{array}\right)$$

$$\mathsf{wfAccSucc}(e, e_p, r, p, (\sigma_v^0, \sigma_v)) \triangleq \sigma_v^0 \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(r) \land \sigma_v^0 \vdash \langle e_p, \sigma_v\rangle \Downarrow_{\mathsf{v}} \mathsf{V}(p)$$

Figure 3.19: Rule for the simulation of `remcheck` $\mathtt{acc}(e.f, e_p)$. The definition of exhAccSucc is given in Figure 3.8 on page 116.

Additional parameterisation for state relations and success predicates in rules

In all the rules presented so far, the different premises of rules modelling simulations of Viper effects use only state relations and success predicates that are parametric w.r.t. their standard input parameters, *i.e.* the Viper and Boogie state for the state relation, and the input and output Viper state of the simulated Viper effect for the success predicate. However, there are cases where it is beneficial or even necessary for state relations and success predicates to depend on further parameters.

As an example, consider the rule racc-sim in Figure 3.19, which decomposes the simulation of `remcheck` $\mathtt{acc}(e.f, e_p)$ into the simulation of three separate Viper effects: (1) the well-definedness check of the receiver e and permission expression e_p (via the wfSimList instantiation from Figure 3.14 on page 141), (2) a check exhAccSucc ensuring that the operation will not fail (from the semantics; see Figure 3.8 on page 116), and (3) the actual update of the Viper state, which removes the permission.

Note that R_B, Succ_A, Succ_B are parameterised by a reference value r and a permission value p, where r and p model the values that the receiver e and the permission expression e_p evaluate to in the cases when e and e_p are well-defined. R_B maps such values to a state relation. Succ_A and Succ_B map such values to a success predicate. r and p are universally quantified in the premises, and constrained by the success predicates. This setup permits in particular R_B to directly talk about the values that e and e_p evaluate to as specified by the success and failure predicates, which makes it make more convenient to prove the premises for concrete Boogie encodings. This is particularly useful for justifying cases where the simulation of the non-failure check establishes a property on r or p, which is then used in the simulation of the state update. For example, the existing Viper-to-Boogie translation stores p into a temporary variable that is used for both the non-failure check and the state update (see line 11 in Figure 3.9 on page 124).

Without this additional parameterisation, the success predicate would existentially quantify over r and p, and an instantiation of the state relation would have to express r and p explicitly via the Viper expression evaluation judgement. As a result, a client of the rule proving the premises for a concrete Boogie encoding would have to use that the expression evaluation is deterministic in order to link the existentially quantified r and p in the success predicate with the corresponding reference and permission values used in the instantiated state relation. Our parameterisation of the state relation avoids this technical overhead, which makes it make more convenient to prove the premises for concrete Boogie encodings.

Let us take a closer look at other parts of the rule. The second premise includes contextual information, namely the conjunct wfAccSucc expressing that e and e_p are well-defined (which is ensured by the first premise) and evaluate to the reference value r and permission value p. The third premise modelling the removal of the permission includes the same conjunct and that the operation will succeed (exhAccSucc). Without the latter, we could not prove in general that the resulting Boogie state satisfies crucial invariants, for instance, that none of the permissions stored in the Boogie state are negative. Again, we are agnostic as to syntactically *how* this is achieved by this check: our rule does *not* require the Boogie program to emit an explicit Boogie **`assert`** command checking that the permission is nonnegative. This is important, since the existing implementation omits such a command, for example, if the permission is expressed via the literal **`write`** (*i.e.* full permission).

The extra parameterisation that we discussed in the simulation rule for **`remcheck`** **`acc`**$(e.f, e_p)$ made the rule more convenient to use but was not technically necessary. However, there are cases where the parameterisation is necessary to express a suitable simulation rule. One such example is the simulation rule for method calls. Before we show this rule, let us first look at a simpler example that captures the essence of this second extra parameterisation use case.

In particular, let us revisit the rule ASSERT-SIM for the simulation of **`assert`** A shown in Figure 3.17 on page 146 for the purposes of illustrating the second extra parameterisation case. In ASSERT-SIM, the second premise's first disjunct must somehow recover the Viper state σ_v^{init} before the **`assert`** operation was executed (*i.e.* before the **`remcheck`** effect modelled in the first premise) in order to express the Viper effect that resets the state back to σ_v^{init}. The first disjunct recovers σ_v^{init} by observing that σ_v^{init} must be the same as the permission definedness state σ_v^0 at the end of the **`remcheck`** effect modelled in the first premise. This observation holds because of the semantics of **`remcheck`**. If this observation did not hold, because, for example, the permission definedness state changed during the reduction of **`remcheck`**, then one would not be able to recover σ_v^{init} in the first disjunct's success predicate without any further adjustments, and thus would not be able to express a precise rule for the simulation of **`assert`** A. How would we write the rule for simulating **`assert`** A in such a case? (For our generation of certificates, we use the original rule ASSERT-SIM for the simulation of **`assert`** A; we are answering this question for the purpose of illustrating the second extra parameterisation case before showing the simulation rule for method calls, which requires this parameterisation.)

Figure 3.20 provides an answer via an alternative rule ASSERT-ALT-SIM for

$$\dfrac{\begin{array}{c}\mathsf{bSim}_{\Gamma_b}(R, \lambda\sigma_v\ \sigma_b.\ (R_A(\sigma_v))(\sigma_v, \sigma_b), \gamma, \gamma_1) \\ \forall\sigma_v^{init}.\ \mathsf{rcSim}_{\Gamma_b}([\lambda(\sigma_v^0, \sigma_v)\ \sigma_b.\ \sigma_v^0 = \sigma_v \wedge (R_A(\sigma_v^{init}))(\sigma_v, \sigma_b)], R_B(\sigma_v^{init}), A, \gamma_1, \gamma_2) \\ \left(\begin{array}{l}(\forall\sigma_v^{init}.\ \mathsf{sim}_{\Gamma_b}(R_B(\sigma_v^{init}), [\lambda(\_, \sigma_v)\ \sigma_b.\ R'(\sigma_v, \sigma_b)], \mathit{Succ}_2(\sigma_v^{init}), \lambda\_.\ \bot, \gamma_2, \gamma')) \vee \\ \mathsf{bSim}_{\Gamma_b}(R, R', \gamma, \gamma')\end{array}\right)\end{array}}{\mathsf{stmSim}_{\Gamma_v,\Gamma_b}(R, R', \texttt{assert}\ A, \gamma, \gamma')}\ (\textsc{assert-alt-sim})$$

$$\mathit{Succ}_2(\sigma_v^{init}) \triangleq \lambda(\_, \sigma_v)\ (\_, \sigma_v').\ \sigma_v' = \sigma_v^{init} \wedge \sigma_v^{init} \vdash \langle A, \sigma_v^{init}\rangle \rightarrow_{\mathsf{rc}} \mathsf{N}(\sigma_v)$$

Figure 3.20: Alternative rule for the simulation of `assert` A. The differences compared to the original rule ASSERT-SIM in Figure 3.17 are highlighted. bSim is defined in in Figure 3.15.

the simulation of `assert`, which does not rely on `remcheck` leaving the permission definedness state unchanged. The key idea in ASSERT-ALT-SIM is to add an additional parameter to the state relation and success predicate where this parameter models the state σ_v^{init} before the `assert` operation. Succ_2 expresses resetting the Viper state in the success predicate simply by stating that the output Viper state σ_v' matches σ_v^{init}; the permission definedness state is completely ignored in the success predicate.

The first premise in ASSERT-ALT-SIM captures the state σ_v^{init} before the `assert` operation in the state relation using R_A, which is a function from the extra parameter state to a state relation. The second premise and the first disjunct of the third premise are analogous to the premises in the original rule. The only difference is the universal quantification over the extra state parameter σ_v^{init}. Intuitively, the universal quantification in the second premise expresses that the simulation must preserve any state that was initially captured in the state relation. The universal quantification in the third premise expresses that for any captured state, the Boogie code must simulate a reset to this state. If R_A and R_B did not have an extra parameter modelling σ_v^{init}, then there would be no way of explicitly capturing σ_v^{init} in instantiations of the state relation, and thus one would not be able to prove the simulation of resetting the Viper state to σ_v^{init} for concrete Boogie encodings.

Let us now turn our attention to the rule MCALL-SIM in Figure 3.21 for the simulation of a method call $zs := m(es)$, which uses a similar extra parameterisation pattern to the above alternative simulation rule for `assert`. At a high level, MCALL-SIM decomposes the simulation into the simulation of the well-definedness of the arguments es ($\mathsf{wfSimList}_{\Gamma_b}(R, R, es, \gamma, \gamma_1)$) and into the simulation of following effects which closely follow the formal semantics definition (see STMT-MCALL in Figure 3.5 on page 113): (S1) changing the Viper store to capture the mapping from the callee's formal arguments to the argument values (modelled by Succ_1), (S2) exhaling the callee's precondition, (S3) nondeterministically assigning values to the target variables (modelled by Succ_3), (S4) inhaling the callee's postcondition, and (S5) resetting the store to the one before the call where the target variables are updated to the nondeterministically assigned values (modelled by Succ_5).

The key point is that expressing the final effect (*i.e.* resetting the store) requires knowledge of the store as it was *before the call*. As a result, the rule parameterises state relations and success predicates in the rule with the extra parameter σ_v^*, which models the state before the call.

$$\dfrac{\begin{array}{c}\mathsf{Methods}(\Gamma_v)(m) = \mathit{mdecl} \\ \mathsf{wfSimList}_{\Gamma_b}(R, R, es, \gamma, \gamma_1) \\ \forall \sigma_v^*\ \sigma_b^*\ vs\ vs'.\ R(\sigma_v^*, \sigma_b^*) \wedge \mathit{Prems}_{\Gamma_v}(\sigma_v^*, \mathit{mdecl}, vs, vs') \Longrightarrow \\ \left(\begin{array}{l}\mathsf{sim}_{\Gamma_b}(\lambda\sigma_v\ \sigma_b.\ (\sigma_v, \sigma_b) = (\sigma_v^*, \sigma_b^*), R_2(\sigma_v^*, \sigma_b^*), \mathit{Succ}_1, \lambda\_.\ \bot, \gamma_1, \gamma_2) \wedge \\ \mathsf{stmSim}_{\Gamma_v,\Gamma_b}(R_2(\sigma_v^*, \sigma_b^*), R_3(\sigma_v^*, \sigma_b^*), \texttt{exhale}\ \mathsf{pre}(\mathit{mdecl}), \gamma_2, \gamma_3) \wedge \\ \mathsf{sim}_{\Gamma_b}(R_3(\sigma_v^*, \sigma_b^*), R_4(\sigma_v^*, \sigma_b^*), \mathit{Succ}_3(\sigma_v^*), \lambda\_.\ \bot, \gamma_3, \gamma_4) \wedge \\ \mathsf{stmSim}_{\Gamma_v,\Gamma_b}(R_4(\sigma_v^*, \sigma_b^*), R_5(\sigma_v^*, \sigma_b^*), \texttt{inhale}\ \mathsf{post}(\mathit{mdecl}), \gamma_4, \gamma_5) \wedge \\ \mathsf{sim}_{\Gamma_b}(R_5(\sigma_v^*, \sigma_b^*), R', \mathit{Succ}_5(\sigma_v^*), \lambda\_.\ \bot, \gamma_5, \gamma')\end{array}\right)\begin{array}{l}\text{(S1)}\\ \text{(S2)}\\ \text{(S3)}\\ \text{(S4)}\\ \text{(S5)}\end{array}\end{array}}{\mathsf{stmSim}_{\Gamma_v,\Gamma_b}(R, R', zs := m(es), \gamma, \gamma')}\ \text{(MCALL-SIM)}$$

$$\mathit{Prems}_{\Gamma_v}(\sigma_v^*, \mathit{mdecl}, vs, vs') \triangleq \left(\begin{array}{l}\sigma_v^* \vdash \langle es, \sigma_v^* \rangle\ [\Downarrow]_{\mathsf{V}}\ \mathsf{V}(vs) \wedge \mathsf{map}(\lambda v.\ \mathsf{typVpr}(v), vs) = \mathsf{argTypes}(\mathit{mdecl}) \wedge \\ \mathsf{map}(\lambda z. \mathsf{Vars}(\Gamma_v)(z), zs) = \mathsf{retTypes}(\mathit{mdecl}) \wedge \\ \mathsf{map}(\lambda v.\ \mathsf{typVpr}(v), vs') = \mathsf{retTypes}(\mathit{mdecl})\end{array}\right)$$

$$\mathit{Succ}_1 \triangleq \lambda\sigma_v\ \sigma_v'.\ \sigma_v' = (xs\ [\mapsto]\ vs, \mathsf{H}(\sigma_v), \Pi(\sigma_v))$$

$$\mathit{Succ}_3(\sigma_v^*) \triangleq \lambda\sigma_v\ \sigma_v'.\ \left(\begin{array}{l}\sigma_v' = (xs@ys\ [\mapsto]\ vs@vs', \mathsf{H}(\sigma_v), \Pi(\sigma_v)) \wedge \\ \Gamma_v \vdash \langle \texttt{exhale}\ \mathsf{pre}(\mathit{mdecl}), (xs\ [\mapsto]\ vs, \mathsf{H}(\sigma_v^*), \Pi(\sigma_v^*))\rangle \rightarrow_{\mathsf{V}} \mathsf{N}(\sigma_v)\end{array}\right)$$

$$\mathit{Succ}_5(\sigma_v^*) \triangleq \lambda\sigma_v\ \sigma_v'.\ \left(\begin{array}{l}\sigma_v' = (\mathsf{ST}(\sigma_v^*)(zs[\mapsto]vs'), \mathsf{H}(\sigma_v), \Pi(\sigma_v)) \wedge \\ (\exists\sigma_v^{pre}.\ \Gamma_v \vdash \langle \texttt{inhale}\ \mathsf{post}(\mathit{mdecl}), (xs@ys\ [\mapsto]\ vs@vs', \mathsf{H}(\sigma_v^{pre}), \Pi(\sigma_v^{pre}))\rangle \rightarrow_{\mathsf{V}} \mathsf{N}(\sigma_v))\end{array}\right)$$

Figure 3.21: Simulation rule for method calls. Here, xs are the formal arguments and ys are the formal target variables in the method declaration mdecl. The term $xs@ys$ denotes the list obtained by appending lists xs and ys. The term $xs[\mapsto]vs$ denotes the mapping where the i-th element in xs is mapped to the i-th element in vs. The term $\mathsf{ST}(\sigma_v^*)(zs[\mapsto]vs')$ is a mapping that maps any element x not in zs to $\mathsf{ST}(\sigma_v^*)(x)$ and maps any element z in zs to $(zs[\mapsto]vs')(z)$.

Additionally, for the convenience of clients proving premises of the rules for concrete Boogie encodings, the rule also adds the Boogie state σ_b^* after the well-definedness check of expressions as an extra parameter for the state relations. This way clients can choose to capture properties in the state relation directly via σ_b^* and prove that this corresponds to capturing a property of σ_v^*. For instance, when applying the rule as part of the certificate generation, we track in the parameterised state relations that the Boogie state maps Boogie variables capturing Viper variables in scope before the call to the same value as σ_b^* (*i.e.* these variables are not affected by the encoding of the call).

Note that as with most of our simulation rules, MCALL-SIM contains contextual information for various of the simulations in the premise. For instance, the effect that nondeterministically assigns values for the target variables (modelled by Succ_3) includes that the prior state can be reached by exhaling the callee's precondition. As we will discuss in Subsection 3.5.3, we use this contextual information to justify why omitting well-definedness checks in the postcondition is fine. Finally, the left-hand side of the rule's third premise provides all the information that one gets for any reduction of the method call where the arguments are well-defined (*e.g.* that the argument values have the types declared in the callee's method signature).

$$\frac{\begin{array}{c}\mathsf{rcInvSim}^{Q}_{\Gamma_b}(R, R', A_1, \gamma, \gamma') \quad \mathsf{rcInvSim}^{Q}_{\Gamma_b}(R', R'', A_2, \gamma', \gamma'') \\ \forall \sigma_v^0, \sigma_v.\ Q(A_1 \mathbin{\&\&} A_2, (\sigma_v^0, \sigma_v)) \Rightarrow \left(\begin{array}{l} Q(A_1, (\sigma_v^0, \sigma_v)) \land \\ \forall \sigma_v'.\ \sigma_v^0 \vdash \langle A_1, \sigma_v \rangle \rightarrow_{\mathsf{rc}} \mathsf{N}(\sigma_v') \Rightarrow Q(A_2, (\sigma_v^0, \sigma_v')) \end{array}\right)\end{array}}{\mathsf{rcInvSim}^{Q}_{\Gamma_v}(R, R'', A_1 \mathbin{\&\&} A_2, \gamma, \gamma'')}\ (\textsc{rsep-sim})$$

$$\mathsf{rcInvSim}^{Q}_{\Gamma_b}(R, R', A, \gamma, \gamma') \triangleq \mathsf{rcSim}_{\Gamma_b}((\lambda \tau\ \sigma_b.\ R(\tau, \sigma_b) \land Q(A, \tau)), R', A, \gamma, \gamma')$$

Figure 3.22: The instantiation for simulating **remcheck** A with assertion predicate Q (bottom) and the corresponding rule for the separating conjunction (top).

3.4.5. Injecting Non-Local Hypotheses into Simulation Proofs

Our rules are designed to be parametric in the state relation between the Viper and Boogie state and permit adjusting this state relation at different points in the simulation proof (*e.g.* via the Boogie propagation rule BPROP in Figure 3.15). In principle, this allows the injection of arbitrary non-locally-justified hypotheses into all of our simulation judgements. However, automating the *usage* of general logical assumptions embedded into our state relations can become a challenge in itself.

For example, the existing Viper-to-Boogie translation omits the well-definedness checks of expressions in the translation of **remcheck** A and **inhale** A in certain cases (as we discussed in Subsection 3.3.3 on page 127). This is justified, because A is checked to be *well-formed* non-locally in those cases, but to *use* this additional hypothesis requires propagating and adjusting the hypothesis throughout the cases of the definitions of **remcheck** A and **inhale** A. As a result, without additional care, the automation of corresponding simulation proofs must explicitly handle these adjustments of the state relation and must deal with a variety of different state relations resulting from these adjustments. This makes automation more challenging.

As a final ingredient of our methodology, we allow *specialised* instantiations of the generic forward simulation judgement sim that encapsulate these extra hypotheses via an *additional* parameter. This allows capturing the property required to propagate the hypothesis *separately* via *additional* premises in simulation rules. As a result, applications of the rule need not adjust the state relation explicitly, and the specialised rule replaces recurring adaptations and proof steps at the level of the state relation by the justification of additional premises in the rules, which simplifies automation.

For example, Figure 3.22 shows (at the bottom) an instantiation rcInvSim of sim that expresses the simulation of **remcheck** A, parameterised with an additional predicate Q on assertions. Its definition in terms of rcSim requires $Q(A, \tau)$ to hold as part of the input state relation, where τ is a pair containing the permission definedness state and reduction state. The specialised rule RSEP-SIM (top of Figure 3.22) for **remcheck** A_1 && A_2 decomposes the simulation into simulations for A_1 and A_2. Both sub-simulations use the *same* predicate Q, such that applications of the rule do *not* need to adjust the state relations explicitly to reflect that, for example, Q holds for A_1 and A_2 in the respective states. This property is ensured *separately* by the third premise. In practice, for a specific Q, we prove the third premise once and for all for all assertions A_1 and A_2, which further simplifies automation. Note that the same parameter can be instantiated in many ways to capture different non-local hypotheses for different applications of the same rule.

Deriving rule RSEP-SIM

The rule RSEP-SIM can be derived from the instantiation-independent composition rule COMP and consequence rule CONSEQ shown in Figure 3.15 on page 143, which shows another example for the usefulness of our instantiation-independent rules. The proof works as follows at a high level: The proof first applies the composition rule COMP where (1) the intermediate state relation is chosen to be the input state relation of the second premise in RSEP-SIM if one unfolds the definition of rcInvSim (that is, $\lambda\tau\ \sigma_b.\ R_2(\tau, \sigma_b) \wedge Q(A_2, \tau)$), and (2) the two pairs of success and failure predicates for the two simulations being composed are chosen to express the simulation of `remcheck` A_1 and `remcheck` A_2, respectively. This application leads to four proof goals (given by the instantiated premises of COMP), where the two proof goals constraining the two pairs of success and failure predicates follow directly from the definition of `remcheck`. The remaining two proof goals express the simulation of `remcheck` A_1 (goal 1) and `remcheck` A_2 (goal 2), respectively. Goal 2 precisely matches the second premise in RSEP-SIM, and thus is trivially proved. Goal 1 does not match the first premise in RSEP-SIM, because both the input and output state relations do not match. To bridge this gap, the proof applies the consequence rule CONSEQ to goal 1, which allows weakening the input relation and strengthening the output relation to reduce goal 1 to the first premise in RSEP-SIM. The weakening and strengthening conditions imposed by CONSEQ are justified by the final premise in RSEP-SIM.

In our formalisation, we express all of our simulation rules directly via rcInvSim instead of rcSim. In the case where we do not need to propagate additional hypotheses via a separate predicate, we instantiate the predicate Q in rcInvSim using the trivial predicate that always holds (*i.e.* $Q(A, (\sigma_v^0, \sigma_v)) = \top$). This way we need only one set of rules, and thus only one certification strategy for simulations of `remcheck`, which leads to a uniform approach. Our certification strategy takes as input some parameters that are instantiated in different ways for different translations of `remcheck` (*e.g.* a parameter specifying how to instantiate Q), but most of the steps in our strategy are independent from these parameters. As a result, we need to maintain only one certification strategy for `remcheck` instead of two completely separate certification strategies for the two translations (*i.e.* the translation that omits well-definedness checks of expressions and the one that does not). This becomes even more important as one must support more alternatives justified by non-local checks or adds support for additional features that must be taken into account by the certification strategy.

Figure 3.23 shows a selection of rules that are all directly defined in terms of rcInvSim. The rule EXH2-SIM for the simulation of `exhale` A is the same as the one we initially showed in Figure 3.16 on page 145, except that now the additional predicate Q is included and there is an additional

$$\dfrac{\begin{array}{c} \forall \sigma_v\ \sigma_b.\ R(\sigma_v, \sigma_b) \Longrightarrow Q(A, (\sigma_v, \sigma_v)) \\ \mathsf{rcInvSim}^{Q}_{\Gamma_b}([\lambda(\sigma_v^0, \sigma_v)\ \sigma_b.\ \sigma_v^0 = \sigma_v \wedge R(\sigma_v, \sigma_b)], R', A, \gamma, \gamma') \quad \text{(sim. of } \texttt{remcheck}\ A) \\ \mathsf{sim}_{\Gamma_b}(R', [\lambda(\_, \sigma_v)\ \sigma_b.\ R''(\sigma_v, \sigma_b)], \mathit{Succ}_2, \lambda\_.\ \bot, \gamma', \gamma'') \quad \text{(non-det. selection)} \end{array}}{\mathsf{stmSim}_{\Gamma_v,\Gamma_b}(R, R'', \texttt{exhale}\ A, \gamma, \gamma'')}\ (\textsc{exh2-sim})$$

$$\dfrac{\begin{array}{c} \mathsf{wfSimList}_{\Gamma_b}(R_{acc}, R_{acc}, [e, e_p], \gamma, \gamma_1) \\ \forall r, p.\ \mathsf{sim}_{\Gamma_b}(R_{acc}, R_B(r,p), \mathit{Succ}_A(r,p), \mathit{Fail}_A(r,p), \gamma_1, \gamma_2) \quad \text{(non-failure check)} \\ \forall r, p.\ \mathsf{sim}_{\Gamma_b}(R_B(r,p), R', \mathit{Succ}_B(r,p), (\lambda\_.\ \bot), \gamma_2, \gamma') \quad \text{(state update)} \end{array}}{\mathsf{rcInvSim}^{Q}_{\Gamma_b}(R, R', \texttt{acc}(e.f, e_p), \gamma, \gamma')}\ (\textsc{racc2-sim})$$

$$R_{acc} \triangleq \lambda\tau\ \sigma_b.\ R(\tau, \sigma_b) \wedge Q(\texttt{acc}(e.f, e_p), \tau)$$

$$\dfrac{\begin{array}{c} \mathsf{wfSim}_{\Gamma_b}(R_{exp}, R_{exp}, e, \gamma, \gamma') \\ \mathsf{sim}_{\Gamma_b}(R_{exp}, R'', \mathit{Succ}_{exp}, \mathit{Fail}_{exp}, \gamma', \gamma'') \quad \text{(expression holds)} \end{array}}{\mathsf{rcInvSim}^{Q}_{\Gamma_v}(R, R'', e, \gamma, \gamma'')}\ (\textsc{rexp-sim})$$

$$R_{exp} \triangleq \lambda\tau\ \sigma_b.\ R(\tau, \sigma_b) \wedge Q(e, \tau)$$

$$\mathit{Succ}_{exp} \triangleq \lambda(\sigma_v^0, \sigma_v)(\sigma_v^1, \sigma_v').\ (\sigma_v^0, \sigma_v) = (\sigma_v^1, \sigma_v') \wedge \sigma_v^0 \vdash \langle e, \sigma_v \rangle \Downarrow_{\mathsf{V}} \mathsf{VBoolVal(true)}$$

$$\mathit{Fail}_{exp} \triangleq \lambda(\sigma_v^0, \sigma_v).\ \sigma_v^0 \vdash \langle e, \sigma_v \rangle \Downarrow_{\mathsf{V}} \mathsf{VBoolVal(false)}$$

$$\dfrac{\begin{array}{c} \mathsf{wfSim}_{\Gamma_b}(R_{imp}, R_{imp}, e, \gamma, ([]; \texttt{if}\ (e_b)\ \{\ b :: bs\ \}\ \texttt{else}\ \{\ [[]; \epsilon]\ \}, \mathbf{KSeq}(b', \mathcal{K}))) \\ \mathsf{expRelVprBoogie}_{\Gamma_b}(R, e, e_b) \\ \mathsf{rcInvSim}^{Q}_{\Gamma_b}(R, R, A, (b, \mathsf{blocksToCont}(bs, \mathbf{KSeq}(b', \mathcal{K}))), (b', \mathcal{K})) \\ \forall \sigma_v^0, \sigma_v.\ Q(e \Rightarrow A, (\sigma_v^0, \sigma_v)) \wedge \sigma_v^0 \vdash \langle e, \sigma_v \rangle \Downarrow_{\mathsf{V}} \mathsf{VBoolVal(true)} \Longrightarrow Q(A, (\sigma_v^0, \sigma_v)) \end{array}}{\mathsf{rcInvSim}^{Q}_{\Gamma_v}(R, R, e \Rightarrow A, \gamma, (b', \mathcal{K}))}\ (\textsc{rimp-sim})$$

$$R_{imp} \triangleq \lambda\tau\ \sigma_b.\ R(\tau, \sigma_b) \wedge Q(e \Rightarrow A, \tau)$$

Figure 3.23: `exhale` simulation rule expressed via rcInvSim and selection of rules for the simulation of `remcheck` using rcInvSim. Succ_A and Succ_B are defined in Figure 3.19. expRelVprBoogie relates a Viper and a Boogie expression; we discuss this relation in more detail in Subsection 3.5.5.

premise requiring that the input state relation implies that the predicate holds. The rule racc2-sim for `remcheck` `acc`$(e.f, e_p)$ is essentially the same as the one that we discussed earlier in Figure 3.19 on page 148 except that the additional predicate parameter Q is made explicit in racc2-sim. The rule rexp-sim for the Boolean expression case is straightforward. The rule rimp-sim for the implication has one high-level difference to the rest of the rules presented so far: the rule exposes some modest implementation details, since the rule forces the Boogie program to encode the Viper implication via an if-statement, where the implication's right-hand side must be simulated in the then-branch. We chose to prove this rule because it was most convenient for our use case. However, it would be straightforward to abstract this implementation detail away as well to support more possible translations (*e.g.* to support a case that omits the if-statement if the left-hand side is known to be true statically). In such a more generic rule, one would add a premise for the case when the left-hand side evaluates to false. Moreover, one would explicitly add the condition specifying that the left-hand side evaluates to true (resp. false) to the state relation in the premise simulating the `remcheck` of the right-hand side (resp. simulating the case when the left-hand side evaluates to false).

We proceed analogously for the simulation of **inhale** A. That is, we use the following instantiation inhInvSim:

$$\begin{aligned}&\mathsf{inhInvSim}^{Q}_{\Gamma_b}(R, R', A, \gamma, \gamma') \triangleq\\&\mathsf{inhSim}_{\Gamma_b}(\lambda \sigma_v\ \sigma_b.\ R(\sigma_v, \sigma_b) \wedge Q(A, \sigma_v), R', A, \gamma, \gamma')\end{aligned}$$

Here, the extra predicate Q takes only one Viper state as input instead of a pair, since the inhale reduction tracks only a single Viper state. As for **remcheck**, we express all of our simulation rules directly via inhInvSim instead of inhSim.

In summary, our methodology solves all three challenges outlined in Section 3.1 and expanded on in Section 3.3. The *large semantic gap* between the input language and the IVL is handled by decomposing the statements of the input language into smaller effects and defining for each of them instantiations of a generic forward simulation relation. The parameterisation of this relation allows us, in particular, to capture information about the context in which the effects are executed. This parameterisation also supports *diverse translations* by abstracting from the details of the translation. Finally, *non-locality* is handled by capturing properties checked elsewhere in the state relations, and by devising specialised rules that simplify the certificate generation. All of these ideas are needed to validate the existing Viper-to-Boogie translation, but apply equally to other front-end translations.

3.5. Putting The Methodology to Work

This section shows how to apply the general methodology that we presented in Section 3.4 to concrete front-end translations in order to automatically generate certificates, which can then be successfully checked by Isabelle automatically . This section does so by presenting key ingredients involved in an application to the existing Viper-to-Boogie translation, but these could also be applied to other front-end translations. In particular, this section presents our instantiation of the state relation connecting Viper with Boogie states (Subsection 3.5.1), how we track auxiliary Boogie variables (Subsection 3.5.2), a concrete instance of non-local reasoning (Subsection 3.5.3), how our proof automation works (Subsection 3.5.4), and how we automatically relate Viper and Boogie expressions (Subsection 3.5.5). Finally, this section shows how to use forward simulation proofs to generate a certificate proving soundness for a concrete Viper program and its Boogie translation (Subsection 3.5.6).

3.5.1. State Relation

In order to use the rules from Section 3.4 for deriving forward simulation judgements for concrete Viper and Boogie constructs, we must instantiate the state relation between Viper and Boogie states. Moreover, the rules from Section 3.4 allow us to adjust state relations as needed during a simulation proof. Thus, it is possible to use different instantiations of the state relation during a simulation proof. We use this flexibility of adjusting the state relations in many ways, *e.g.* when (1) a scoped

$$\begin{aligned}
&\mathsf{SR}^{Tr,AV}_{\Gamma_v,\Gamma_b}((\sigma^0_v,\sigma_v),\sigma_b) \triangleq \mathsf{consistent}(\sigma^0_v) \wedge \mathsf{consistent}(\sigma_v) \wedge \mathsf{ST}(\sigma^0_v) = \mathsf{ST}(\sigma_v) \wedge \mathsf{H}(\sigma^0_v) = \mathsf{H}(\sigma_v) \wedge \\
&\mathsf{stRel}_{\Gamma_b}(\mathsf{var}(Tr),\sigma_v,\sigma_b) \wedge \mathsf{hmRel}_{\Gamma_v,\Gamma_b}(\mathsf{field}(Tr),\mathsf{H}(Tr),\mathsf{M}(Tr),\sigma_v,\sigma_b) \wedge \\
&\mathsf{hmRel}_{\Gamma_v,\Gamma_b}(\mathsf{field}(Tr),\mathsf{H}^0(Tr),\mathsf{M}^0(Tr),\sigma^0_v,\sigma_b) \wedge \\
&\mathsf{fieldRel}_{\Gamma_v,\Gamma_b}(\mathsf{field}(Tr),\sigma_b) \wedge \mathsf{constRep}_{\Gamma_b}(\mathsf{const}(Tr),\sigma_b) \wedge (\forall x,P.\ AV(x) = P \Longrightarrow P(\sigma_b(x))) \wedge \\
&\mathsf{stateWellTy}(\mathsf{TypeInterp}(\Gamma_b),\mathsf{Vars}(\Gamma_b),\emptyset,\sigma_b) \wedge \\
&\mathsf{disjointList}([\{\mathsf{H}(Tr),\mathsf{H}^0(Tr)\},\{\mathsf{M}(Tr),\mathsf{M}^0(Tr)\},\mathsf{ran}(\mathsf{var}(Tr)),\mathsf{ran}(\mathsf{field}(Tr)),\mathsf{range}(\mathsf{const}(Tr)),\mathsf{dom}(AV)])
\end{aligned}$$

Figure 3.24: A simplified version of our instantiated state relation between Viper and Boogie states. disjointList ensures that all the sets in a given list are pairwise disjoint. dom provides the domain of a partial function. range and ran provide the range of a total and partial function, respectively.

Viper variable is introduced, (2) a new auxiliary Boogie variable is introduced, (3) the Boogie variables tracking the Viper state are changed. To have a systematic way of instantiating the state relation and facilitating proof automation for handling state relation adjustments, we build in a stylised form for expressing state relations via two parameters. The first parameter is a partial *auxiliary variable map* from auxiliary Boogie variables to *predicates* on Boogie values (mappings from Boogie values to Booleans) stating what logical condition holds for the value stored for an auxiliary Boogie variable (we will discuss concrete predicates for auxiliary variables in Subsection 3.5.2). The second parameter is a *translation record* specifying how key Viper components are represented in the Boogie state. The scenarios above are handled by adjusting one of these two parameters as we will show with concrete examples in later parts of this section.

Translation records comprise: (1) a mapping var(Tr) from Viper variables to their Boogie counterparts, (2) the Boogie variables H(Tr) and M(Tr) representing the Viper heap and permission mask, respectively (and whenever we use a separate permission definedness state, the Boogie variables $\mathsf{H}^0(Tr)$ and $\mathsf{M}^0(Tr)$ representing the corresponding heap and permission mask, respectively), (3) a mapping field(Tr) from Viper fields to corresponding Boogie constants, and (4) a mapping const(Tr) from expected constant identifiers to Boogie constants as represented in the Boogie program.[17]

17: In the existing Viper-to-Boogie translation for our supported Viper subset, there are expected Boogie constant identifiers for the empty (0) permission value, the full (1) permission value, the null reference, and the Boogie value representing the empty permission mask.

Figure 3.24 shows a simplified version of our state relation instantiation SR for translation record Tr, auxiliary variable map AV, Viper context Γ_v, and Boogie context Γ_b, where σ_v and σ_b are the Viper and Boogie states, and σ^0_v is a distinguished Viper permission definedness state (if there is none, then $\sigma_v = \sigma^0_v$). The first line ensures that the Viper states are consistent (*i.e.* there is at most 1 permission to each heap location) and that the permission definedness state and the standard Viper state differ at most on the permission mask. We use consistency, for example, to justify why the `assume GoodMask(M)` Boogie command discussed in Subsection 3.3.6 on page 132 executes normally instead of going to magic (`GoodMask(M)` essentially requires the Viper permission mask to be consistent due to a corresponding axiom). The second and third lines ensure that the Boogie state correctly captures the Viper store (stRel), Viper permission mask, and Viper heap (hmRel).

The next two lines ensure the remaining properties on the Boogie state that we require for our generated certificates: (1) Viper fields are represented correctly in the Boogie state (fieldRel), (2) expected Boogie constants are correctly reflected in the Boogie state (constRep), which means, for example, that the Boogie constant representing the full permission value actually stores the value 1, (3) for each auxiliary variable-predicate pair (x, P) in the auxiliary variable map, P holds for the value stored in the Boogie state for x, and (4) the Boogie state is well-typed. We require the well-typedness of the Boogie state to prove that certain Boogie expressions reduce (using our type soundness result for Boogie expressions discussed in Subsection 2.3.6 on page 35 in Chapter 2).

The final line ensures that Boogie variables modelling different aspects of the encoding must be different. For example, the Boogie variable $\mathtt{H}(\mathit{Tr})$ representing the Viper heap should be different from any of the auxiliary variables in the auxiliary variable map. The benefit of having this property in the state relation is the following. If the Boogie program updates a variable x modelling one aspect in the state relation (via a Boogie assignment), then we directly know that the conditions required by the state relation must still hold for the variables modelling different aspects than x. For example, if the Boogie program updates $\mathtt{H}(\mathit{Tr})$ (*e.g.* to reflect a Viper field assignment), then we know that all the predicates for the tracked auxiliary variables must still hold without requiring further checks. If one did not include the final line in the state relation, then one would have to prove on every such update that the updated variable is indeed different from all the variables tracking different aspects in the state relation.

3.5.2. Dealing with Auxiliary Boogie Variables

We track Boogie variables explicitly in the auxiliary variable map of the state relation whenever we need to track Boogie variables that are not captured by any other part of the state relation. One such example is in the existing Viper-to-Boogie translation for **remcheck acc**$(e.f, e_p)$. Here, the translated Boogie program stores the permission that is to be removed into a temporary variable `tmp` and then uses `tmp` at different points later. We track `tmp` in the auxiliary variable map in order to justify the points where `tmp` is used. See an example translation of **remcheck acc**$(e.f, e_p)$ on lines 10-19 in Figure 3.9 on page 124, where `tmp` is set on line 11.

More concretely, in the simulation proof of **remcheck acc**$(e.f, e_p)$, we adjust the state relation from $\mathsf{SR}^{\mathit{Tr},\mathit{AV}}_{\Gamma_v,\Gamma_b}$ to $\mathsf{SR}^{\mathit{Tr},\mathit{AV}(\mathtt{tmp}\mapsto\lambda v.\ v=\mathsf{RealVal}(p))}_{\Gamma_v,\Gamma_b}$ as a result of the Boogie assignment initialising `tmp`, where p is the real value that e_p evaluates to.[18] In subsequent steps of the simulation proof, we can then use the fact that `tmp` stores p. Once `tmp` is not used any more, we revert the state relation back to $\mathsf{SR}^{\mathit{Tr},\mathit{AV}}_{\Gamma_v,\Gamma_b}$.

18: Recall that we can define the state relation in terms of p directly, since our corresponding simulation rule RACC-SIM in Figure 3.19 on page 148 parameterises the state relation with p.

A different example where we track variables in the auxiliary variable map is to justify the translation of **assert** A. Recall that the existing Viper-to-Boogie translation of **assert** A works as follows (see the Boogie encoding shown on the far left of Figure 3.10 on page 126): the current Boogie variables `H` and `M` tracking the Viper heap and permission mask are copied into unused Boogie variables `aH` and `aM`. Then, the **remcheck** A operation is encoded w.r.t. `aH` and `aM` (leaving `H` and `M` unchanged), and at the end, the encoding continues with the original variables `H` and

`M`. To justify the encoding of **`assert`** A, we need to prove that `H` and `M` are indeed not modified by the encoding of **`remcheck`** A such that we can then prove that `H` and `M` model the correct (unchanged) heap and permission mask *after* **`assert`** A. We achieve this by tracking `H` and `M` in the auxiliary variable map during the simulation proof of **`remcheck`** A.

More concretely, in the simulation proof of **`remcheck`** A, as a result of the two assignments to `aH` and `aM`, we adjust the state relation from $\mathsf{SR}^{Tr,AV}_{\Gamma_v,\Gamma_b}$ to:[19]

19: Note that here `H` and `M` correspond to $\mathsf{H}(Tr)$ and $\mathsf{M}(Tr)$, respectively.

$$\begin{aligned}
&\lambda(\sigma_v^0, \sigma_v)\ \sigma_b.\ \mathsf{SR}^{Tr',AV'}_{\Gamma_v,\Gamma_b}((\sigma_v^0, \sigma_v), \sigma_b) \text{ where}\\
&\quad Tr' \triangleq Tr(\mathtt{H} \mapsto \mathtt{aH}, \mathtt{M} \mapsto \mathtt{aM})\\
&\quad AV' \triangleq \left(\begin{array}{l} AV(\mathtt{H} \mapsto \lambda h_b.\ \mathsf{heapRel}_{\Gamma_b}(\mathsf{field}(Tr), \mathtt{H}, \mathsf{H}(\sigma_v^0), h_b))\\ \quad(\mathtt{M} \mapsto \lambda m_b.\ \mathsf{maskRel}_{\Gamma_b}(\mathsf{field}(Tr), \mathtt{M}, \Pi(\sigma_v^0), m_b)) \end{array}\right)
\end{aligned}$$

So, the translation record is updated to reflect `aH` and `aM` to be the new variables modelling the Viper heap and permission mask. Moreover, the updated auxiliary variable map AV' states that `H` and `M` are related to the Viper heap and permission mask of the permission definedness state σ_v^0 ($\mathsf{heapRel}$ and $\mathsf{maskRel}$ are defined accordingly). As a result, since the permission definedness state remains the same during the reduction of **`remcheck`** A, the adjusted state relation tells us at the end of the simulation of **`remcheck`** that `H` and `M` indeed still model the Viper state *before* the **`assert`** statement. Therefore, after the simulation proof of **`remcheck`** A, we can revert the state relation back to $\mathsf{SR}^{Tr,AV}_{\Gamma_v,\Gamma_b}$ as part of the Viper state reset effect modelled by the first disjunct in the second premise of the rule ASSERT-SIM in Figure 3.17 on page 146.[20]

20: The Viper state reset effect in ASSERT-SIM explicitly states that the state is reset to the permission definedness state and thus we do not need to explicitly ensure during the simulation proof that the permission definedness state stays the same. The **`remcheck`** simulation (rcSim) already captures that the permission definedness state stays the same in the success predicate.

In our simulation proofs for method calls, we also use the auxiliary variable map to prove that certain variables are not modified during subsimulations (similarly to our approach for **`assert`**). In particular, our method call rule MCALL-SIM (see Figure 3.21 on page 151) requires us to prove simulations w.r.t. a Viper state whose store tracks only the formal argument variables and formal target variables of the callee's method declaration; the final simulation premise in the rule then resets the Viper store to the store before the call where the target variables are adjusted accordingly. So, during the simulation proofs that deal with Viper states tracking only the formal argument and formal target variables, we must explicitly ensure that the Boogie variables qs modelling the Viper store *before the call* are not modified. We achieve this by updating the auxiliary variable map to include all variables qs before the call, where the corresponding predicate for a variable q in qs is that the value of q in the current Boogie state is identical to the corresponding value in the Boogie state σ_b^* reached right before the code simulating the method call (the rule MCALL-SIM provides σ_b^* as a parameter to the intermediate state relations).

Syntactic check for unmodified variables

As we showed, one use case for the auxiliary variable map is to prove that certain variables are not modified during the simulation of some Viper effect. It would be more convenient if we could instead just syntactically check that the Boogie code simulating the Viper effect does not modify the variables in question. This would avoid the need to track these variables explicitly in the state relation during the Viper effect. Doing so is not easily possible in our current setup because of how the internal Boogie AST representation is structured, which we discuss as part of future work in Subsection 3.9.4 on page 200.

3.5.3. Non-Locality

For most translations of **`remcheck`** A and **`inhale`** A, the existing Viper-to-Boogie translation generates well-definedness checks in the Boogie program corresponding to expressions evaluated in A. However, as discussed in Subsection 3.3.3 on page 127, specifically when translating the **`remcheck`** of a method call's precondition (as part of an **`exhale`**) and the **`inhale`** of a method call's postcondition, the existing translation omits these well-definedness checks for the corresponding **`remcheck`** and **`inhale`** operations. This optimisation is justified by a *non-local check*: the Boogie code for the *callee*'s translation checks that the callee's specification is *well-formed* (see Definition 3.3.1 on page 129 for the specification well-formedness definition), which implies that expressions evaluated during the corresponding **`remcheck`** and **`inhale`** operations must be well-defined. In Subsection 3.3.4 on page 128, we explained at a high level why this non-local check justifies the optimisation performed by the existing Viper-to-Boogie translation. In this subsection, we will make this justification formal by showing how we establish the simulation of the **`remcheck`** and **`inhale`** operations as part of calls.

Our standard simulation proof for **`remcheck`** A and **`inhale`** A would fail if we did not reflect the consequences of this non-local guarantee, since our standard simulation proof would expect expressions evaluated in A to be checked to be well-defined. So, we must reflect the consequences of this non-local guarantee *in a way that is used automatically during the proof*. We instantiate the general strategy outlined in Subsection 3.4.5 on page 152 for this purpose, which allows us to choose a predicate Q_{pre} (resp. Q_{post}) on assertions (these predicates are functions from assertions and Viper states to Booleans) that will be applied throughout the simulation proof for **`remcheck`** A (resp. **`inhale`** A). The idea is that during the simulation proof we get the guarantee provided by Q_{pre} in addition to the state relation between Viper and Boogie states. We choose Q_{pre} (resp. Q_{post}) such that they capture the non-local guarantee in a way that can be propagated through the simulation proofs for the **`remcheck`** A operation (resp. **`inhale`** A), and such that they guarantee that expressions evaluated during these operations must be well-defined. In the following, we first discuss our approach for **`remcheck`** and then discuss our approach for **`inhale`**.

The `remcheck` case

Our general strategy in Subsection 3.4.5 requires choosing Q_{pre} such that $Q_{pre}(A, (\sigma_v^0, \sigma_v))$ satisfies the following properties (A is an assertion, σ_v^0 is the permission definedness state, and σ_v is the reduction state):

- (Property 1): $Q_{pre}(A, (\sigma_v, \sigma_v))$ must be implied by the non-local check and the state relation before the **remcheck** operation (*i.e.* where A is the callee's precondition and σ_v is the state right before the **remcheck** operation); this property is reflected in the first premise of rule EXH2-SIM in Figure 3.23 on page 154.[21]
- (Property 2): $Q_{pre}(A, (\sigma_v^0, \sigma_v))$ must imply that every definedness subexpression e of A is well-defined (*i.e.* $\neg(\sigma_v^0 \vdash \langle e, \sigma_v\rangle \Downarrow_\mathsf{v} \lightning)$).[22]
- (Property 3): $Q_{pre}(A, (\sigma_v^0, \sigma_v))$ must be *preserved* for subassertions of A that are reduced as part of the **remcheck** operation. By preservation we mean the following: If $Q_{pre}(A, (\sigma_v^0, \sigma_v))$ holds, then if **remcheck** A' (where A' is a direct subassertion of A) is reduced in reduction state σ_v' as part of the reduction of **remcheck** A in permission definedness state σ_v^0 and reduction state σ_v, then $Q(A', (\sigma_v^0, \sigma_v'))$ must hold; the third premise of RSEP-SIM in Figure 3.22 on page 152 expresses this property for the separating conjunction.

21: Note that in rule EXH2-SIM, the non-local guarantee must be embedded into the state relation to satisfy the first premise.

22: The function defineSubExprsA in Figure 3.6 on page 114 expresses the definedness subexpressions of an assertion; if a definedness subexpression of an assertion is ill-defined in a state, then the corresponding **inhale** and **exhale** fails in that state.

Note that (Property 2) and (Property 3) together ensure that $Q_{pre}(A, (\sigma_v^0, \sigma_v))$ implies that any expression evaluated in the reduction of **remcheck** A must be well-defined.

We instantiate Q_{pre} with the following definition, which satisfies the three properties:

$$Q_{pre}(A, \sigma_v^0, \sigma_v) \triangleq \mathsf{consistent}(\sigma_v^0) \land \exists \sigma_v^i.\ \sigma_v \oplus \sigma_v^i \preceq \sigma_v^0 \land \neg\langle A, \sigma_v^i\rangle \longrightarrow_{\mathsf{inh}} \mathsf{F}$$

Recall that $\langle A, \sigma_v^i\rangle \longrightarrow_{\mathsf{inh}} r_v$ expresses the reduction of **inhale** A in state σ_v^i to outcome r_v. The symbol $\oplus$ is an addition operator that results in a state whose permission mask is the pointwise sum of the two states and whose store and heap is the same as in the input states; $\oplus$ is a partial operation that is defined iff the two input states have the same store and heap. $\sigma_v \preceq \sigma_v'$ holds iff the permission mask of σ_v' is pointwise larger than the permission mask of σ_v, and the heap and store are identical.[23]

23: The formal definition is $\sigma_v \preceq \sigma_v' \triangleq \exists \sigma_v''.\ \sigma_v \oplus \sigma_v'' = \sigma_v'$.

Our instantiation for Q_{pre} states three conditions. First, the permission definedness state must be consistent, which clearly is always the case, since our semantics ensures that every reached state is consistent (since the initial state is consistent). Second, there must exist some state σ_v^i containing at most the permission difference between the reduction state and permission definedness state. Third, an **inhale** starting from σ_v^i cannot fail.

Let us see why our instantiation for Q_{pre} satisfies the three required properties, which will also provide an intuition for the instantiation itself. Let us consider (Property 1) first. The well-formedness of the callee's specification (see Definition 3.3.1 on page 129), which is guaranteed by the non-local check, ensures that an **inhale** of the precondition A_{pre} does not fail starting from any state. Moreover, since our usual state relation (Figure 3.24 on page 156) ensures that the Viper state σ_v is consistent, we get that our usual state relation and the non-local check together imply $Q_{pre}(A_{pre}, (\sigma_v, \sigma_v))$ for an empty σ_v^i (*i.e.* no permissions in σ_v^i).[24] Thus, (Property 1) holds.

24: More concretely, in our proof, we make sure that in our use of the method call rule MCALL-SIM in Figure 3.21, our instantiation of R_2 guarantees that $R_2(\sigma_v^*, \sigma_b^*)(\sigma_v, \sigma_b)$ implies $Q_{pre}(A, (\sigma_v, \sigma_v))$ where A is the callee's precondition, which then allows us to establish the first premise of rule EXH2-SIM in Figure 3.23 on page 154.

Let us consider (Property 2) next. From our instantiation of Q_{pre} we know that there is a state σ_v^i such that an **inhale** A from σ_v^i never fails. Thus, we get that every definedness subexpression e of A is well-defined in σ_v^i (*i.e.*

$\neg(\sigma_v^i \vdash \langle e, \sigma_v^i\rangle \Downarrow_{\mathsf{v}} \lightning))$, because otherwise `inhale` A would fail in σ_v^i due to the rule INH-SUBEXP-FAIL in Figure 3.7 on page 115. From this, we conclude that (Property 2) holds. That is, every definedness subexpression e of A is well-defined in state σ_v with permission definedness state σ_v^0 (*i.e.* $\neg(\sigma_v^0 \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \lightning))$, because (1) σ_v^0 has at least as many permissions as σ_v^i, and (2) σ_v^i and σ_v agree on the heap and store, and the evaluation of subexpressions of specifications do not depend on the permission mask in σ_v.[25]

25: The evaluation of e in σ_v with permission definedness state σ_v^0 depends on the permission mask in σ_v iff e contains permission introspection. While we currently do not support permission introspection in our generated certificates, the Viper language disallows permission introspection in method specifications except if *inhale-exhale assertions* are used, which are not part of our Viper subset.

Showing (Property 3) is the most challenging part, namely showing that Q_{pre} is preserved for direct subassertions of A that are reduced as part of the `remcheck` operation. This is especially challenging for the separating conjunction. For the separating conjunction we must show that if $Q_{pre}(A_1 \ \&\& \ A_2, \sigma_v^0, \sigma_v)$ holds, then (C1) $Q_{pre}(A_1, \sigma_v^0, \sigma_v)$ holds, and (C2) if `remcheck` A_1 reduces successfully from the permission definedness state σ_v^0 and reduction state σ_v to outcome $\mathsf{N}(\sigma_v')$, then $Q_{pre}(A_2, \sigma_v^0, \sigma_v')$ holds. (C1) follows directly, since `inhale` A_1 cannot fail if `inhale` A_1 && A_2 cannot fail. (C2) is the challenging part. To prove (C2), we require the following technical lemma stating a partial *inversion* property between `remcheck` and `inhale`:

Lemma 3.5.1 *Let A be an assertion without permission introspection and let σ_v^0, σ_v', σ_v^i, σ_v^s be Viper states, where $\sigma_v^s = \sigma_v^i \oplus (\sigma_v \ominus \sigma_v')$ and σ_v^s is consistent. If $\sigma_v^0 \vdash \langle A, \sigma_v\rangle \longrightarrow_{\mathsf{rc}} \mathsf{N}(\sigma_v')$ and $\neg\langle A, \sigma_v^i\rangle \longrightarrow_{\mathsf{inh}} \mathsf{F}$ holds, then $\langle A, \sigma_v^i\rangle \longrightarrow_{\mathsf{inh}} \mathsf{N}(\sigma_v^s)$.*

In this lemma, $\ominus$ is the pointwise subtraction of the permission masks (leaving the heap and store unchanged); $\ominus$ is a partial operation that is defined iff the two input states have same the store and heap and the subtraction yields nonnegative permission amounts.[26]

26: In our formalisation, $\ominus$ is not partial, because we reuse a general library for *partial commutative monoids* that defines $\ominus$ in terms of $\oplus$ in a total way such that $\ominus$ precisely matches our description if the two input states have the same store and heap, and the subtraction yields nonnegative permission amounts. Since we use $\ominus$ only whenever this condition indeed holds, our lemmas and proofs work both with the partial version presented here and the total version used in the formalisation.

The lemma essentially states that the permissions removed by a successful `remcheck` A operation (expressed by $\sigma_v \ominus \sigma_v'$) are exactly those that will be added by a corresponding (non-failing) `inhale` A operation. Moreover, the lemma provides sufficient conditions for a non-failing `inhale` A operation to *not* go to magic. Intuitively, the latter is the case because (1) the fact that `remcheck` A succeeds tells us that the logical constraints in A are satisfied, and (2) adding the removed permissions to σ_v^i does not yield an inconsistent state (since the lemma requires σ_v^s to be consistent). We prove this lemma by induction on the reduction of `remcheck`.

We discuss why the lemma considers only assertions without permission introspection later. First, we discuss the intuition for how this lemma connects to our instantiation of Q_{pre} and the proof of (C2). The existentially quantified state σ_v^i in Q_{pre} represents the permissions that the ongoing `remcheck` operation *has removed so far*. So, for the proof of (C2) the lemma helps us follows: Suppose we know $Q_{pre}(A_1 \ \&\& \ A_2, \sigma_v^0, \sigma_v)$ holds before the `remcheck` A_1 && A_2 operation where σ_v^* is a witness for the existentially quantified state in Q_{pre}. Then, the idea is to use the lemma to justify $Q_{pre}(A_2, \sigma_v^0, \cdot)$ after a successful `remcheck` A_1 operation with a witness that contains all the permissions in σ_v^* *in addition to* the permission removed by `remcheck` A_1. The fact that the witness for the existentially quantified state in Q_{pre} represents the permissions removed so far guarantees that adding the permission in the reduction state σ_v to this witness yields at most the permission in the permission definedness state σ_v^0 (*i.e.* the condition $\sigma_v \oplus \sigma_v^i \preceq \sigma_v^0$ in Q_{pre} always holds). This is because the permission definedness state contains precisely the permission at the beginning of the `remcheck` operation, and thus this operation cannot remove more permission than contained in the permission definedness state. In particular, the sum $\sigma_v \oplus \sigma_v^i$ stays the same throughout the proof for different witnesses for σ_v^i (since the permissions removed from σ_v are transferred to the witness of σ_v^i).

Now that we have an intuition for Lemma 3.5.1 and its use in combination with Q_{pre}, let us consider a proof of (C2). Assume (a) $Q_{pre}(A_1 \;\&\&\; A_2, \sigma_v^0, \sigma_v)$ and (b) $\sigma_v^0 \vdash \langle A_1, \sigma_v\rangle \longrightarrow_{\mathsf{rc}} \mathsf{N}(\sigma_v')$ holds. Let σ_v^* be a witness for the existentially quantified state in (a). We must show $Q_{pre}(A_2, \sigma_v^0, \sigma_v')$, which we aim to do using witness $\sigma_v^s \triangleq \sigma_v^* \oplus (\sigma_v \ominus \sigma_v')$. We can show the first conjunct within the existential quantifier in $Q_{pre}(A_2, \sigma_v^0, \sigma_v')$ (*i.e.* $\sigma_v' \oplus \sigma_v^s \preceq \sigma_v^0$) by using that $\sigma_v' \oplus \sigma_v^s = \sigma_v \oplus \sigma_v^* \preceq \sigma_v^0$ (we get $\sigma_v \oplus \sigma_v^* \preceq \sigma_v^0$ from (a)). That means, as discussed above, the sum of the reduction state and the witness for the existentially quantified state stays the same for the instantiation that we assume (*i.e.* (a)) and the instantiation that we want to prove.

For the second conjunct within the existential quantifier in $Q_{pre}(A_2, \sigma_v^0, \sigma_v')$ (*i.e.* $\neg\langle A_2, \sigma_v^s\rangle \longrightarrow_{\mathsf{inh}} \mathsf{F}$), we want to apply Lemma 3.5.1, which requires establishing the following premises of the lemma (the others are trivially guaranteed from our assumptions): inhaling A_1 from σ_v^* does not fail, and σ_v^s is consistent. From (a) we know that there is a state σ_v^* from which inhaling $A_1 \;\&\&\; A_2$ cannot fail. As a result, inhaling A_1 from this state cannot fail either. Moreover, σ_v^s must be consistent because we know that $\sigma_v^s \preceq \sigma_v^0$ (since $\sigma_v' \oplus \sigma_v^s \preceq \sigma_v^0$ as we established above). Thus, since from (a) we also know that σ_v^0 is consistent, we get that σ_v^s must be consistent (consistency is downward monotonic).

Therefore, we can use Lemma 3.5.1 to get that inhaling A_1 from σ_v^* results in outcome $\mathsf{N}(\sigma_v^s)$. Using this, we can complete the proof of (C2) by establishing the second conjunct within the existential quantifier of $Q_{pre}(A_2, \sigma_v^0, \sigma_v')$. The reason is that we know that inhaling A_2 from σ_v^s cannot fail, since we know from (a) that inhaling $A_1 \;\&\&\; A_2$ from σ_v^* never fails, and we know that inhaling A_1 from σ_v^* results in outcome $\mathsf{N}(\sigma_v^s)$. So, if inhaling A_2 from σ_v^s failed, then inhaling $A_1 \;\&\&\; A_2$ from σ_v^* would fail (due to rule INH-SEP-N in Figure 3.7 on page 115), which cannot be the case. This concludes the proof of (C2).

Finally, note that the lemma requires that the assertion has no permission introspection, since σ_v^i and σ_v in general have different permission masks and the `remcheck` and `inhale` operations manipulate the permissions differently. This is not an issue for our use case, since permission introspection is not permitted in Viper method specifications (except in *inhale-exhale assertions*, which we do not support). Nevertheless, to see why the lemma does not hold in general if permission introspection were permitted, consider the following assertion A_{perm}:

```
acc(x.f, write) &&
(perm(x.f) == write ==> acc(y.g, write))
```

Executing `remcheck` A_{perm} in a state σ_v with full permission to `x.f` and no other permission succeeds and results in a state σ_v' without any permissions. This is because `perm(x.f)` in the implication's left-hand side evaluates to 0 (since the first conjunct removed the permission) and thus the right-hand side is not evaluated. However, executing `inhale` A_{perm} in a state σ_v^i with no permissions results in a normal outcome whose state σ_v^j has full permission to *both* `x.f` *and* `y.g`. This is because here `perm(x.f)` in the implication's left-hand side evaluates to full permission (since the first conjunct added the permission). Thus, σ_v^j does not match $\sigma_v^i \oplus (\sigma_v \ominus \sigma_v')$, which would contain permission only to `x.f` but not `y.g`. As a result, the lemma would not hold.

The `inhale` case

Now let us briefly turn to justifying translations of inhaling a method call's postcondition, which omit well-definedness checks. Analogously to the `remcheck` case, we use the general strategy in Subsection 3.4.5, which requires us to instantiate $Q_{post}(A, \sigma_v)$ such that the following properties hold: (1) $Q_{post}(\mathsf{post}(m), \sigma_v)$ holds for σ_v right before the `inhale`, (2) the state relation and $Q_{post}(A, \sigma_v)$ imply that every definedness subexpression e of A is well-defined (*i.e.* $\neg(\sigma_v \vdash \langle e, \sigma_v\rangle \Downarrow_{\mathsf{v}} \lightning)$), and (3) Q_{post} must be preserved for subassertions of A that are reduced as part of the `inhale` operation. Note that properties (2) and (3) together ensure that $Q_{post}(A, \sigma_v)$ implies that any expression evaluated in the reduction of `inhale` A must be well-defined.

We instantiate Q_{post} with the following definition, which satisfies the three properties:

$$Q_{post}(A, \sigma_v) \triangleq \neg(\langle A, \sigma_v\rangle \longrightarrow_{\mathsf{inh}} \mathsf{F})$$

Let us see why the three properties are guaranteed for this instantiation. First, the well-formedness of the callee's specification (see Definition 3.3.1 on page 129) guaranteed by the non-local check ensures that inhaling the postcondition cannot fail from any consistent state whose store is compatible with the callee's precondition. From our usual state relation (Figure 3.24) we know that the postcondition is inhaled from a consistent state. Moreover, since the postcondition is inhaled only *after* the precondition was successfully exhaled, we can prove that the postcondition is inhaled only in states whose store is compatible with the callee's precondition. For this proof, we again use Lemma 3.5.1 to connect an `exhale` of the precondition with an `inhale` of the precondition, since the well-formedness definition uses the latter to express compatibility of the store with the callee's precondition. Thus, the first property holds.[27]

27: More concretely, in our application of the method call rule MCALL-SIM in Figure 3.21, we choose R_4 such that $R_4(\sigma_v^*, \sigma_b^*)(\sigma_v, \sigma_b)$ implies $Q_{post}(A, \sigma_v)$, where A is the callee's postcondition. This is possible because a corresponding success predicate in the rule includes the condition expressing that the state in which the postcondition is inhaled can be reached via an `exhale` of the precondition. This allows us to conclude that the `inhale` occurs in a state whose store is compatible with the precondition.

It is easy to see why the second and third property are satisfied by our instantiation. The second property (every definedness subexpression of A is well-defined) follows directly from the fact that an `inhale` fails if a definedness subexpression is ill-defined (rule INH-SUBEXP-FAIL). The third property also follows directly from the semantics of `inhale`. Note that the reason why the third property is much easier to show here than in the `remcheck` case is that here the actual operation is an `inhale` and the instantiation of Q_{post} is *also* expressed via an `inhale`, while in the `remcheck` case the two are expressed via different operations (`remcheck` and `inhale`).

$$\text{Proof Tree } T_1 : \quad \dfrac{\dfrac{\boxed{\text{Proof } \mathscr{P}_2 \text{ (hint 3)}}}{\mathsf{rcSim}_{\Gamma_b}(R_2, R_2, \texttt{acc}(\texttt{x.f}, \texttt{q}), \gamma_1, \gamma_2)} \qquad \dfrac{\boxed{\text{Proof } \mathscr{P}_3 \text{ (no hint)}}}{\mathsf{rcSim}_{\Gamma_b}(R_2, R_2, \texttt{y.g > x.f}, \gamma_2, \gamma_3)}}{\mathsf{rcInvSim}^{Q}_{\Gamma_b}(R_2, R_2, \texttt{acc}(\texttt{x.f}, \texttt{q}) \ \&\& \ \texttt{y.g > x.f}, \gamma_1, \gamma_3)} \ (\textsc{rsep-sim})$$

$$\text{Proof Tree } T_2 : \quad \dfrac{\dfrac{}{\forall \tau \ \sigma_b \ . \ R_2(\tau, \sigma_b) \Longrightarrow R_3(\tau, \sigma_b)} \qquad \dfrac{\boxed{\text{Proof } \mathscr{P}_4 \text{ (hints 4 and 5)}}}{\mathsf{sim}_{\Gamma_b}(R_3, R_3, Succ_2, \lambda\_. \ \bot, \gamma_3, \gamma')}}{\mathsf{sim}_{\Gamma_b}(R_2, R_3, Succ_2, \lambda\_. \ \bot, \gamma_3, \gamma')} \ (\textsc{weaken-input})$$

$$\dfrac{\dfrac{\dfrac{\boxed{\text{Proof } \mathscr{P}_1 \text{ (hint 2)}}}{\mathsf{bSim}_{\Gamma_b}(R_1, R_2, \gamma, \gamma_1)} \qquad (\text{Proof Tree } T_1)}{\mathsf{rcInvSim}^{Q}_{\Gamma_b}(R_1, R_2, \texttt{acc}(\texttt{x.f}, \texttt{q}) \ \&\& \ \texttt{y.g > x.f}, \gamma, \gamma_3)} \ (\textsc{rcprop}) \qquad (\text{Proof Tree } T_2)}{\mathsf{stmSim}_{\Gamma_v, \Gamma_b}(R, R, \texttt{exhale acc}(\texttt{x.f}, \texttt{q}) \ \&\& \ \texttt{y.g > x.f}, \gamma, \gamma')} \ \underbrace{(\textsc{exh2-sim})}_{\text{hint 1}}$$

$$R \triangleq \lambda(\sigma_v, \sigma_b). \ \mathsf{SR}^{Tr, AV}_{\Gamma_v, \Gamma_b}((\sigma_v, \sigma_v), \sigma_b) \qquad Q(A, (\sigma^0_v, \sigma_v)) = \top$$

$$R_1 \triangleq \lambda(\sigma^0_v, \sigma_v) \ \sigma_b. \ \sigma^0_v = \sigma_v \wedge R(\sigma_v, \sigma_b)$$

$$R_2 \triangleq \mathsf{SR}^{Tr_1, AV}_{\Gamma_v, \Gamma_b} \qquad Tr_1 \triangleq Tr(\mathsf{M}^0 \mapsto \texttt{WM})$$

$$R_3 \triangleq \lambda(\_, \sigma_v) \ \sigma_b. \ R(\sigma_v, \sigma_b)$$

Figure 3.25: Proof tree constructed by our proof automation for the simulation of `exhale acc(x.f, q) && y.g > x.f` via the Boogie statement in Figure 3.9 on page 124 shown on lines 10-23. The automation uses generated hints for the application of rule EXH2-SIM, and for proofs at the leaves ($\mathscr{P}_i$; left abstract here). The Boogie program points γ, γ_1, γ_2, γ_3, and γ' are the points in Figure 3.9 on page 124 starting on lines 10, 11, 17, 20, and 23, respectively. SR is our state relation instantiation introduced in Subsection 3.5.1 on page 155. $Succ_2$ is defined in Figure 3.16 on page 145 (where the assertion is A_1 && A_2). The extra premises for predicate Q in the applications of rules EXH2-SIM and RSEP-SIM are not shown; they trivially hold for the chosen Q here. Rules RCPROP and WEAKEN-INPUT are derived from BPROP and CONSEQ (Figure 3.15 on page 143), respectively.

3.5.4. Proof Automation

We have extended the Viper-to-Boogie implementation to automatically generate an Isabelle certificate relating the source Viper and target Boogie program for a given run. The generated certificates contain sufficient information such that Isabelle can automatically check them successfully. To make this automatic generation and subsequent checking possible, we instrument fewer than 500 lines of the existing implementation to produce *hints*, which provide extra information about the Boogie encoding. A core component of our proof automation is an Isabelle tactic that uses these hints to automatically prove forward simulations. The tactic applies the rules provided by our methodology (Section 3.4) to decompose simulations into smaller ones and generates proofs for *atomic simulations* that are not further decomposed. Atomic simulations could technically be decomposed even further, but since these simulations are simple enough, our tactic does not further decompose them. Our instrumentation generates two kinds of hints for the tactic: (1) hints indicating which candidate of multiple diverse translations is used, and (2) hints specifying how to instantiate parameters and discharge premises of a rule. The generated hints contain sufficient information to enable automation.

As a concrete example, consider Figure 3.25, which shows the proof generated by our tactic (represented via a proof tree) for the forward simulation of `exhale acc(x.f, q) && y.g > x.f` via the Boogie statement on lines 10-23 in Figure 3.9 on page 124. Hints 1 and 4 in Figure 3.25 are hints of the first kind. Hint 1 specifies that well-definedness checks are not omitted in the translation of `remcheck`; as a result, the tactic instantiates predicate Q in the rcInvSim simulation with the trivial predicate that always holds (see Subsection 3.4.5 on page 152 for the purpose of this predicate in general). Hint 4 specifies that the nondeterministic heap assignment is not omitted in the Boogie code (see Subsection 3.3.2 on page 126 for when it is omitted), which directs the tactic to use a specific rule (not shown in the figure). Hints 2, 3, and 5 in Figure 3.25 are hints of the second kind. Each of them provides information on temporary Boogie variables (name and lemma showing the declared type is the expected one) in Figure 3.9. The temporary variables here are (1) `WM` to set up the permission definedness state on line 10 (hint 2), which results in a change of the translation record (see Subsection 3.5.1 on page 155) in R_2, (2) `tmp` to store the permission on line 11 (hint 3), which is used to adjust the auxiliary variable map (see Subsection 3.5.1) in proof $\mathcal{P}_2$, and (3) `H'` to perform the nondeterministic selection on line 20 (hint 5).

After decomposing the simulation, our tactic must automatically prove the atomic simulations. In Figure 3.25, $\mathcal{P}_1$ and $\mathcal{P}_4$ are such proofs. $\mathcal{P}_2$ and $\mathcal{P}_3$ further decompose the simulation before reaching atomic simulations ($\mathcal{P}_2$ does so via the rule RACC2-SIM shown in Figure 3.23 on page 154). We use two main automation approaches for atomic simulations. Firstly, we prove (once and for all) simple lemmas about our state relation instantiation (see Subsection 3.5.1) and about the behaviours of small sequences of simple Boogie commands; these are applied (and their hypotheses discharged) automatically when needed. These lemmas are used for only small parts of the overall translation. Secondly, we prove (once and for all) simulation rules that capture effects simulated by Boogie `assume` and `assert` commands for arbitrary expressions. This generality enables a tactic to automatically prove Viper effects that are simulated via a combination of these two commands. Both of these automation approaches rely on a tactic that automatically shows a Viper expression and a corresponding Boogie expression are related (we will discuss the relation of expressions in Subsection 3.5.5).

Our tactic uses both of these approaches for the example in Figure 3.25. Proof $\mathcal{P}_2$ uses the second approach for justifying the nonfailure check for `remcheck acc`$(e.f, p)$ shown on lines 11-15 in Figure 3.9.[28] Proofs $\mathcal{P}_1$ and $\mathcal{P}_4$ use the first approach. Moreover, $\mathcal{P}_2$ also uses the first approach for dealing with the assignment to the temporary Boogie variable storing the to-be-removed permission amount and the assignment updating the Boogie variable modelling the Viper permission mask. In the following, we make the two automation approaches more concrete.

28: The approach is designed to work without any changes to the tactic even if the expressions in the two `assert` statements were changed to be syntactically different.

First automation approach: lemmas on the state relation instantiation and on sequences of Boogie commands

For dealing with the assignment to a temporary Boogie variable, $\mathcal{P}_2$ uses the following lemma that we proved once and for all:

Lemma 3.5.2 *Let the following be arbitrary: Γ_b (Boogie context) ,Tr (translation record), AV (auxiliary variable map), σ_v^0 and σ_v (Viper states), σ_b (Boogie state), x (Boogie variable), e_v (Viper expression), e_b (Boogie expression), v (Viper value). If*

1. $\mathit{SR}^{\mathit{Tr},\mathit{AV}}_{\Gamma_v,\Gamma_b}((\sigma_v^0,\sigma_v),\sigma_b)$
2. $\sigma_v^0 \vdash \langle e_v,\sigma_v\rangle \Downarrow_{\mathsf{V}} V(v)$
3. $\mathit{expRelVprBoogie}_{\Gamma_b}(\mathit{SR}^{\mathit{Tr},\mathit{AV}}_{\Gamma_v,\Gamma_b}, e_v, e_b)$
4. *x is not included in the Boogie variables tracked in Tr and AV*
5. $\mathsf{lookup}_T(\mathit{Vars}(\Gamma_b),x) = \mathsf{typ}_{\mathit{TypeInterp}(\Gamma_b)}(\mathsf{vprToBoogieVal}(v))$

holds, then there is a Boogie state σ_b' such that the following conditions hold:

1. $\Gamma_b \vdash ((x := e_b \ :: cs; ctrl, \mathcal{K}), N(\sigma_b)) \rightarrow^*_{\mathsf{AST2}} ((cs; ctrl, \mathcal{K}), N(\sigma_b'))$
2. $\mathit{SR}^{\mathit{Tr},\mathit{AV}'}_{\Gamma_v,\Gamma_b}((\sigma_v^0,\sigma_v),\sigma_b')$, *where AV′ is given by* $\mathit{AV}(x \mapsto \lambda v_b.\ v_b = \mathsf{vprToBoogieVal}(v))$

This lemma's premises includes (1) the relation between a Viper and Boogie state before the assignment via our concrete instantiation SR, (2) that a Viper expression is known to evaluate to some value v ($\sigma_v^0 \vdash \langle e_v,\sigma_v\rangle \Downarrow_{\mathsf{V}} \mathsf{V}(v)$), and (3) that a Boogie expression e_b is related to e_v (this relation is expressed via expRelVprBoogie, which will be discussed in Subsection 3.5.5). The lemma's conclusion states that assigning e_b to Boogie variable x ensures that the Boogie state σ_b' after the Boogie assignment is still related to the Viper state. In this state relation after the assignment, the auxiliary variable map additionally tracks the fact that x holds the Boogie value related to the Viper value v. The Boogie variable's declared type must match the type of the value that the assignment's right-hand side evaluates to (expressed via $\mathsf{vprToBoogieVal}(v)$ in the final premise), because Boogie's assignment reduces only if the assigned value has the declared type. Moreover, x should not be tracked by the translation record and auxiliary variable map in the premise, since otherwise the assignment may violate some condition tracked by the state relation.

As a second example for a lemma used in the first automation approach, consider the following lemma proved once and for all, which is used as part of proof $\mathcal{P}_4$:

Lemma 3.5.3 *Let the following be arbitrary: Γ_b (Boogie context), Tr (translation record), AV (auxiliary variable map), σ_v (Viper state), σ_b (Boogie state), (h,h',m) (Boogie variables), f (function name), $\hat{f}$ (semantic function), $\mathcal{K}$ (Boogie continuation). If*

1. $\mathit{SR}^{\mathit{Tr},\mathit{AV}}_{\Gamma_v,\Gamma_b}((\sigma_v,\sigma_v),\sigma_b)$
2. $\mathit{nonDet}(\sigma_v^0,\sigma_v,\sigma_v')$
3. $h = \mathsf{H}(\mathit{Tr}) \wedge m = \mathsf{M}(\mathit{Tr}) \wedge \mathsf{H}(\mathit{Tr}) = \mathsf{H}^0(\mathit{Tr})$
4. *h' is not included in the Boogie variables tracked in Tr and AV*
5. $\mathsf{lookup}_T(\mathit{Vars}(\Gamma_b),h') = \mathit{HeapType}$
6. $\mathit{FunInterp}(\Gamma_b)(f) = \hat{f}$ *and for any Boogie values $\hat{h},\hat{h}'$ modelling the heap and $\hat{m}$ modelling the mask, $\hat{f}(\hat{h},\hat{h}',\hat{m})$ evaluates to a Boolean value that is true iff $\hat{h}$ and $\hat{h}'$ agree on all reference-field pairs where $\hat{m}$ stores strictly positive permission.*

then there is a Viper state σ_b' such that $\mathit{SR}^{\mathit{Tr},\mathit{AV}}_{\Gamma_v,\Gamma_b}((\sigma_v',\sigma_v'),\sigma_b')$ *and* $\Gamma_b \vdash ((\texttt{havoc}\ h' :: \texttt{assume}\ f(h,h',m) :: h := h' :: \vec{c}; ctrl, \mathcal{K}), N(\sigma_b)) \rightarrow^*_{\mathsf{AST2}}$

$$\dfrac{\begin{array}{c}\forall \sigma_v\ \sigma_v'\ \sigma_b.\ \left(\begin{array}{l}[R(\sigma_v,\sigma_b) \land (Succ(\sigma_v,\sigma_v') \lor Fail(\sigma_v))] \Longrightarrow \\ \Gamma_b,\emptyset \vdash \langle e,\sigma_b\rangle \Downarrow \mathsf{BoolVal}(b(\sigma_v))\end{array}\right) \\ \forall \sigma_v\ \sigma_v'\ \sigma_b.\ [R(\sigma_v,\sigma_b) \land Succ(\sigma_v,\sigma_v')] \Longrightarrow b(\sigma_v) \\ \mathsf{sim}_{\Gamma_b}(R,R',Succ,\lambda\sigma_v.\ Fail(\sigma_v) \land b(\sigma_v),(cs;ctrl,\mathcal{K}),\gamma')\end{array}}{\mathsf{sim}_{\Gamma_b}(R,R',Succ,Fail,(\texttt{assert}\ e :: cs;ctrl,\mathcal{K}),\gamma')}\ \text{(BASSERT-PROP)}$$

$$\dfrac{\begin{array}{c}\forall \sigma_v\ \sigma_v'\ \sigma_b.\ \left(\begin{array}{l}[R(\sigma_v,\sigma_b) \land (Succ(\sigma_v,\sigma_v') \lor Fail(\sigma_v))] \Longrightarrow \\ \Gamma_b,\emptyset \vdash \langle e,\sigma_b\rangle \Downarrow \mathsf{BoolVal}(b(\sigma_v)) \land b(\sigma_v)\end{array}\right) \\ \mathsf{sim}_{\Gamma_b}(R,R',Succ,Fail,(cs;ctrl,\mathcal{K}),\gamma')\end{array}}{\mathsf{sim}_{\Gamma_b}(R,R',Succ,Fail,(\texttt{assume}\ e :: cs;ctrl,\mathcal{K}),\gamma')}\ \text{(BASSUME-PROP)}$$

Figure 3.26: Generic simulation rules for propagating Boogie `assert` and `assume` commands.

$((\vec{c};ctrl,\mathcal{K}),N(\sigma_b'))$ *holds.*

This lemma captures that a `havoc`-`assume`-assignment sequence simulates a nondeterministic heap assignment w.r.t. our state relation instantiation (see Subsection 3.5.1 on page 155).[29] Note that the final premise requires f's interpretation to satisfy a specific property. We apply the lemma only using Boogie contexts with a Boogie function interpretation that satisfies this property. In particular, for the existing Viper-to-Boogie translation, the function f is given by `idOnPositive` presented in Subsection 3.3.6 on page 132; a corresponding Boogie axiom constrains `idOnPositive` in the same way as required by this lemma (see Subsection 3.3.6). We perform all of our simulation proofs with a Boogie function and type interpretation under which the Boogie axioms in the target Boogie program hold, and thus always use an interpretation of `idOnPositive` that satisfies the required property.

29: If the implementation changed the translation for the nondeterministic heap assignment, then we would have to adjust only the tactic's proof strategy for this assignment via a new lemma (*i.e.* proof $\mathcal{P}_4$ in Figure 3.25); the rest would remain unchanged.

Second automation approach: `assert` and `assume` commands

The Boogie encoding employs `assert` commands at multiple points to explicitly reflect conditions imposed by corresponding Viper operations. Moreover, the encoding employs `assume` commands to explicitly reflect conditions ensured by corresponding Viper operations. With our second automation approach, we want to justify these commands automatically for arbitrary syntactic Boolean Boogie expressions. This means in particular, when justifying an `assert` e (resp. `assume` e) command, the automation should work for syntactically different expressions e that semantically evaluate to the same value. For instance, in Figure 3.9 on page 124, the Boogie encoding contains the assertion `assert M[x,f ] >= temp`, which checks whether the permission mask (modelled by Boogie variable `M`) has sufficient permission. We have designed our proof automation to work irrespective of what syntactic expression is used to express whether there is sufficient permission (*e.g.* `temp <= M[x,f]` would be another option). Such automation provides a systematic way of dealing with `assert` and `assume` commands, which will help with extending our certification work to a larger Viper subset.

To achieve this goal, our second proof automation approach uses the generic simulation rules shown in Figure 3.26 to handle simulations of Viper effects where the next Boogie command is an `assert` or `assume`. These simulation rules are applicable for any Boolean expression used in the commands. Our automation applies rule BASSERT-PROP in the case of a Boogie `assert` command and rule BASSUME-PROP in the case of a Boogie `assume` command. Next, we discuss BASSERT-PROP in more detail; the ideas for BASSUME-PROP are similar.

The main intuition of the rule BASSERT-PROP is the following: If one can express the value that the Boogie expression in the `assert` evaluates to in terms of the Viper state, then it becomes feasible to automatically relate the `assert` with the to-be-simulated Viper effect. The first premise in the rule reflects the evaluation of the Boogie expression in terms of the Viper state via a function b from Viper states to Booleans, where all Viper and Boogie states relevant for the simulation are considered. We discuss below how to automatically pick b. The second premise requires that the `assert` does not fail in case the Viper effect succeeds (expressed via the function b), otherwise the simulation does not hold. Proving this second premise automatically is feasible precisely because the `assert` condition is expressed directly via the Viper state. Finally, the third premise expresses the simulation if the `assert` command *succeeds*. Here, the failure predicate explicitly gets the condition that the `assert` succeeds; the success predicate *Succ* already implies the success of the `assert` due to the second premise. Note that if the to-be-simulated Viper effect in the rule's conclusion fails (*i.e.* $\mathit{Fail}(\sigma_v)$ holds), then either the `assert` command fails or succeeds. If the `assert` fails, then the simulation of the (failing) Viper effect is established. If the `assert` succeeds, then the final premise ensures that there is a failing Boogie execution in the *remaining* code.

Our proof automation technique is able to automatically prove the simulation of the nonfailure check for `remcheck acc`$(e.f, p)$ by applying the rule BASSERT-PROP for both `assert` commands emitted in the Boogie encoding (see lines 11-15 in Figure 3.9 on page 124 for an example). After the second application of BASSERT-PROP, the remaining simulation's failure predicate contains (1) the condition defining when `remcheck acc`$(e.f, p)$ fails, and (2) the conditions defining when both `assert` commands succeed. Our automation shows via the application of a built-in Isabelle tactic that (1) and (2) contradict each other, which means that at least one of the `assert` commands fails if `remcheck acc`$(e.f, p)$ fails, which establishes the simulation.

There is still the question of how our approach automatically chooses the function b from Viper states to Booleans in the first premise of BASSERT-PROP, where $b(\sigma_v)$ denotes the value that the `assert` condition e evaluates to in Viper state σ_v. At a high level, our automation chooses b as follows. In a first step, our automation proves evaluation results in terms of the Viper state for all minimal subexpressions in e whose evaluation must consider the state relation between the Viper and Boogie states in order to express the result in terms of the Viper state. In a second step, our automation chooses b by proving the first premise of BASSERT-PROP using the first step's results. Importantly, our automation does not choose b *before* proving the premise. Instead, our automation does the proof while leaving b symbolic. At the end of the proof, a single proof goal remains, which constrains $b(\sigma_v)$ in a way that then allows our automation to choose b in a straightforward way. This second step that chooses b as part of the proof is enabled by our routine use of *schematic variables* in Isabelle (*evars* in other tools), for postponing the choice of witnesses for existentially-quantified values. That is, when beginning the proof of the first premise, b is a schematic variable and by the end of the proof b is correctly instantiated. Our automation uses built-in Isabelle tactics, which can deal with schematic variables and can instantiate them correctly in the final proof goal. The second step proves the first premise by repeatedly applying the Boogie expression evaluation rules (Figure 2.4 on page 29 in Chapter 2) on the first premise of BASSERT-PROP and whenever a subgoal is reached that specifies the evaluation of a minimal subexpression from the previous step, our automation uses the previously computed result. For subgoals obtained from this repeated application that do not involve the evaluation of expressions, our automation uses built-in Isabelle tactics to discharge them.

To illustrate these two steps, suppose that the **`assert`** condition e is `M[x,f] >= temp` as for the sufficient permission check. Here, the minimal subexpressions considered in the first step are `M[x,f]` and `temp`. Our automation shows that `M[x,f]` evaluates to $\Pi(\sigma_v)(x_v, f_v)$, where σ_v is the Viper state, and x_v and f_v are the Viper counterparts of Boogie variables `x` and `f`. Moreover, our automation shows that `temp` evaluates to the permission p that is to be removed. In the second step, our automation applies the Boogie expression evaluation rule for binary operators, and then proves the evaluation of the two operands with the results computed by the first step. The final subgoal requires showing the result of the binary operation, which our automation proves using a built-in Isabelle tactic, which establishes that the result is $\Pi(\sigma_v)(x_v, f_v) \geq p$. As a consequence of proving this final subgoal, the function b, which is a schematic variable before this final proof, is automatically chosen by the built-in Isabelle tactic applied by our automation to be $\lambda\sigma_v.\ \Pi(\sigma_v)(x_v, f_v) \geq p$.

Proving the evaluation of minimal subexpressions

The first step of our automation approach computes the evaluation of minimal subexpressions where one must take the state relation into account. Since we know what each emitted **`assert`** command encodes, we know what minimal subexpressions to expect. Currently, these minimal subexpressions are hardcoded into the tactic for the first step. Thus, if the relevant minimal subexpressions change for an **`assert`** emitted by the translation, then we would have to adjust our tactic. An alternative would be to drive the first step via hints generated by the instrumentation, which would provide information on the relevant minimal subexpressions. Further note that, in the first step, our automation uses simple once-and-for-all proved lemmas for certain parts that can be reused in many contexts. For instance, we prove a generic once-and-for-all proved lemma showing that Boogie expressions of the form `M[e,f]` evaluate to the permission value stored in the corresponding Viper permission mask under certain conditions (*e.g.* if the state relation holds).

Dealing with the structural AST mismatch

A general challenge when our automation applies the rules from Section 3.4 is that the Viper and Boogie ASTs are structured differently (see Subsection 3.2.1). Thus, the automatic selection of Boogie program points in the premises of rules is not immediate. For example, when applying rule EXH2-SIM in Figure 3.25, the tactic cannot easily choose the intermediate program point γ_3 by inspecting the initial program point γ. Instead, the tactic starts proving the first premise with an *existentially quantified* γ_3. Once the proof reaches the goal of proof $\mathcal{P}_1$ (*i.e.* the first atomic simulation), it becomes clear how to advance the program point γ and, by the end of the proof of the first premise of EXH2-SIM, the choice of γ_3 becomes clear. This strategy is enabled by our routine use of schematic variables in Isabelle (evars in other tools), for postponing the choice of witnesses for existentially-quantified values.

3.5.5. Relating Viper and Boogie Expressions

Automatically relating a Viper expression with a corresponding Boogie expression is an important building block of our proof automation. We express the relation between a Viper expression e_v and a Boogie expression e_b w.r.t. Boogie context Γ_b and state relation R using the judgement $\mathsf{expRelVprBoogie}_{\Gamma_b}(R, e_v, e_b)$. Formally, we define the relation between expressions as follows:[30]

30: Here, the state relation R relates a pair of Viper states (permission definedness state and evaluation state) with a Boogie state.

Definition 3.5.1 (Judgement relating a Viper and a Boogie expression)

$$\mathit{expRelVprBoogie}_{\Gamma_b}(R, e_v, e_b) \triangleq$$
$$\forall \sigma_v^0\ \sigma_v^1\ \sigma_v\ \sigma_b\ v.\ [R((\sigma_v^0, \sigma_v), \sigma_b) \land \sigma_v^1 \vdash \langle e_v, \sigma_v \rangle \Downarrow_{\mathsf{V}} \mathsf{V}(v)] \Longrightarrow$$
$$\Gamma_b, \emptyset \vdash \langle e_b, \sigma_b \rangle \Downarrow \mathsf{vprToBoogieVal}(v)$$

The definition essentially expresses that for any related Viper and Boogie states, if the Viper expression evaluates normally to a value v, then the Boogie expression evaluates to the corresponding value (expressed via $\mathsf{vprToBoogieVal}(v)$). There is one technicality in the definition. The permission definedness state σ_v^1 considered for the evaluation of the Viper expression does not necessarily match the permission definedness state σ_v^0 used as input for the state relation. We currently do not require the definition in cases where the states do not match. Nevertheless, we formalise this more general definition where they could differ, because we wanted to leave the option open of further parameterising the expression evaluation judgement such that one could additionally obtain a *total* evaluation. A total evaluation would always yield a value even for ill-defined expressions (*e.g.*, permissions would not be checked for field accesses, and division by 0 would get some fixed value) and thus does not depend on the permission definedness state. Prior formalisations of implicit dynamic frames have used such a total evaluation [107], and so we thought that additionally having a total evaluation would be useful for larger Viper subsets (extensibility to larger Viper subsets being one of our

[107]: Summers et al. (2013), *A Formal Semantics for Isorecursive and Equirecursive State Abstractions*

$$\frac{\forall \sigma_v\ \sigma_v^0\ \sigma_b.\ R((\sigma_v^0,\sigma_v),\sigma_b) \Longrightarrow \begin{pmatrix} \exists w.\ \mathsf{ST}(\sigma_v)(x_v) = w \land \\ \mathsf{lookup}(\mathsf{Vars}(\Gamma_b),\sigma_b,x_b) = \mathsf{vprToBoogieVal}(w) \end{pmatrix}}{\mathsf{expRelVprBoogie}_{\Gamma_b}(R, x_v, x_b)}\ (\textsc{var-rel})$$

$$\frac{\begin{array}{c} bop_v \in \{\,||\,, \&\&, \Rightarrow\} \\ \forall \sigma_v\ \sigma_v^0\ \sigma_b.\ R((\sigma_v^0,\sigma_v),\sigma_b) \Longrightarrow \exists c.\ \Gamma_b,\emptyset \vdash \langle e_b', \sigma_b\rangle \Downarrow \mathsf{BoolVal}(c) \\ \mathsf{expRelVprBoogie}_{\Gamma_b}(R, e_v, e_b) \\ \mathsf{expRelVprBoogie}_{\Gamma_b}(R, e_v', e_b') \end{array}}{\mathsf{expRelVprBoogie}_{\Gamma_b}(R, e_v\ bop_v\ e_v', e_b\ \widehat{bop_v}\ e_b')}\ (\textsc{binop-lazy-rel})$$

$$\frac{\begin{array}{c} \forall \sigma_v\ \sigma_v^0\ \sigma_b.\ R((\sigma_v^0,\sigma_v),\sigma_b) \Longrightarrow \mathsf{SR}^{Tr,AV}_{\Gamma_v,\Gamma_b}((\sigma_v^0,\sigma_v),\sigma_b) \\ \mathsf{Fields}(\Gamma_v)(f_v) = \tau_v \quad f_b = \mathsf{field}(Tr)(f_v) \\ e_b = \mathit{hread}(\mathsf{H}(Tr), e_b', f_b, \mathsf{vprToBoogieTyp}(\tau_v)) \\ \mathsf{heapReadWf}_{\Gamma_b}(\mathit{hread}) \\ \mathsf{expRelVprBoogie}_{\Gamma_b}(R, e_v', e_b') \end{array}}{\mathsf{expRelVprBoogie}_{\Gamma_b}(R, e_v'.f_v, e_b)}\ (\textsc{field-rel})$$

$$\begin{array}{l} \mathsf{heapReadWf}_{\Gamma_b}(\mathit{hread}) \triangleq \forall e_h, e_r, h, r, f, \sigma_b, v, \tau. \\ \begin{pmatrix} \Gamma_b,\emptyset \vdash \langle e_h, \sigma_b\rangle \Downarrow \mathsf{AHeap}(h) \land h(r, \mathsf{NormalField}(f,\tau)) = v \land \\ \mathsf{typ}_{\mathsf{TypeInterp}(\Gamma_b)}(\mathsf{AHeap}(h)) = \mathtt{HeapType} \land \\ \Gamma_b,\emptyset \vdash \langle e_r, \sigma_b\rangle \Downarrow \mathsf{ARef}(r) \land \\ \Gamma_b,\emptyset \vdash \langle e_f, \sigma_b\rangle \Downarrow \mathsf{AField}(\mathsf{NormalField}(f,\tau)) \end{pmatrix} \\ \quad \Longrightarrow \\ \Gamma_b,\emptyset \vdash \langle \mathit{hread}(e_h, e_r, e_f, \mathsf{vprToBoogieTyp}(\tau)), \sigma_b\rangle \Downarrow v \end{array}$$

Figure 3.27: Selected rules for relating a Viper expression with a Boogie expression. The term $\widehat{bop_v}$ denotes the Boogie binary operator corresponding to bop_v. The definition of $\mathsf{heapReadWf}_{\Gamma_b}(\mathit{hread})$ uses constructors for the instantiation of Boogie's abstract value carrier type, which were presented in Subsection 3.3.7 on page 136. This definition expresses that if the parameters of the function *hread* have the expected types, then *hread* returns the value in the input Boogie heap h (a parameter of *hread*) stored at the corresponding heap location (also parameters of *hread*).

design goals). This definition makes explicit that the specific permission definedness state used in the expression evaluation does not matter for this definition, and thus this definition could be made compatible (*i.e.* the result would hold for concrete Viper and Boogie expressions) with a total evaluation. The reason the specific permission definedness state does not matter is that the definition considers only *well-defined* expressions as indicated by the evaluation to a normal value instead of failure. Since the permission definedness state does not influence the *value* that a *well-defined* expression evaluates to, the Boogie expression should evaluate to the corresponding Boogie value irrespective of the permission definedness state used for the Viper evaluation. This definition would be compatible with a total evaluation if one ensures that the ill-defined cases (*e.g.* division by 0) evaluate the same way in Viper and Boogie.

To automate proofs of this expression relation judgement, we prove rules for the different Viper expression constructors that decomposes the proof into proofs relating subexpressions and basic subgoals. Our automation applies decomposition rules as long as possible and handles the remaining subgoals via tactics. Our rules are kept generic for the most part (*e.g.* most of the rules are fully parametric in the state relation) in order to be applicable in different contexts. Both of these points (decomposition and genericity) are similar to how we set up our simulation rules (Section 3.4). A key difference to the simulation rule decomposition is that here the decompositions follow solely the syntactic structure of the expressions as opposed to defining semantic decompositions that go beyond the syntax.

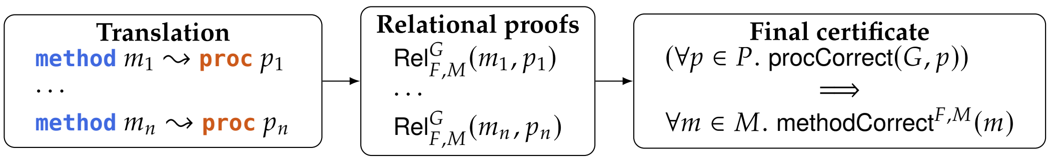
Figure 3.27 shows a selection of rules that we use. Rule VAR-REL relates a Viper variable with a Boogie variable. Its premise states that for related Viper and Boogie states, the Viper store and the Boogie state map the variables to corresponding values. Rule BINOP-LAZY-REL relates a lazy Viper binary operation with a corresponding Boogie operation.[31] The two final premises relate the subexpressions. Compared to Viper, Boogie treats every binary operation *eagerly*, which means in Boogie both operands are always evaluated. As a result, the evaluation of a Viper binary operation does not on its own imply that the corresponding Boogie binary operation reduces to a corresponding value, since the second Boogie operand may be ill-typed. To account for this mismatch, the second premise explicitly requires that the second Boogie operand always reduces to a Boolean value (since all lazy Viper operators are Boolean operators). Our automation proves this second premise by showing that the second Boogie operand is well-typed and then uses Boogie's type soundness for expressions to prove the premise (see Subsection 2.3.6 on page 35 in Chapter 2). The reduction of the first Boogie operand e_b follows from the fact that the first Viper operand e_v is always evaluated and since the premises require that e_v and e_b are related.

31: A binary operation is lazy if the second operand may not be evaluated, since the first operand determines the result.

The most involved rule in Figure 3.27 is FIELD-REL for relating a Viper field access to a corresponding Boogie expression. Here, a difference to the other rules is that the first premise requires that the state relation parameter R implies our state relation instantiation with some translation record Tr and some auxiliary variable map AV (see Subsection 3.5.1 on page 155). This allows the rule to use properties of the translation record such as, for instance, the Viper heap being related to $\mathsf{H}(Tr)$ and $\mathsf{field}(Tr)$ mapping Viper fields to related Boogie constants. While it would be possible to formalise a rule that makes all these properties explicit in the rule without using the state relation instantiation, it would clutter the rule substantially. So, we instead decided to formalise a more specific rule that is more concise.

Let us take a look at the other premises of FIELD-REL. The second premise states that the accessed Viper field indeed exists and stores values of type τ_v. The third premise states that f_b is the Boogie constant modelling the accessed Viper field. The fourth premise states that the Boogie expression e_b representing the field access must depend only on the Boogie variable tracking the Viper heap ($\mathsf{H}(Tr)$), some Boogie receiver expression (e'_b), the Boogie constant related to the Viper field (f_b), and the Boogie type representing values stored in the Viper field ($\mathsf{vprToBoogieTyp}(\tau_v)$). This premise is expressed via a function *hread* that depends only on these parameters, and which can be chosen by a client of the rule.[32] The fifth premise requires that expressions mapped to by *hread* read the expected value from the Boogie heap, which is expressed via $\mathsf{heapReadWf}_{\Gamma_b}(hread)$. The final premise requires that the Viper and Boogie receiver expressions are related.

32: *hread* is not a Boogie function. It is a mapping from Boogie expressions and a Boogie type to a Boogie expression. When using the rule, we instantiate *hread* to map its arguments to a Boogie function call (using the Boogie function `readHeap` shown in Figure 3.13 on page 135).


Translation
method $m_1 \rightsquigarrow$ proc p_1
$\cdots$
method $m_n \rightsquigarrow$ proc p_n
Relational proofs
$\mathsf{Rel}^G_{F,M}(m_1, p_1)$
$\cdots$
$\mathsf{Rel}^G_{F,M}(m_n, p_n)$
Final certificate
$(\forall p \in P.\ \mathsf{procCorrect}(G, p))$
$\Longrightarrow$
$\forall m \in M.\ \mathsf{methodCorrect}^{F,M}(m)$

$$M = \{m_1, m_2, \ldots, m_n\} \qquad P = \{p_1, \ldots, p_n\}$$

$$\mathsf{Rel}^G_{F,M}(m, p) \triangleq \mathsf{procCorrect}(G, p) \Rightarrow \underbrace{\mathsf{SpecWf}^F(m)}_{\text{(C1)}} \wedge \underbrace{\left[(\forall m' \in M.\ \mathsf{SpecWf}^F(m')) \Rightarrow \mathsf{methodCorrect}^{F,M}(m)\right]}_{\text{(C2)}}$$

Figure 3.28: Proof strategy for validating the existing Viper-to-Boogie translation. First, a proof is generated relating each Viper method with the corresponding Boogie procedure. Second, the final certificate is deduced. F denotes the Viper fields, M denotes the Viper methods, G denotes the constants, global variables, Boogie axioms, and functions emitted by the translation. procCorrect is defined in Definition 2.3.1 on page 32, SpecWf is defined in Definition 3.3.1 on page 129, and methodCorrect is defined in Definition 3.2.1 on page 119.

3.5.6. Generating a Certificate of the Final Theorem

We will now discuss, given a Viper program and its Boogie translation, how forward simulation proofs can be used to generate a certificate of the final theorem justifying the soundness of the translation: *i.e.* that the correctness of the Boogie program (*i.e.* the correctness of all contained Boogie procedures) implies the correctness of the Viper program (*i.e.* the correctness of all contained Viper methods).

We decompose the certificate of the final theorem into smaller parts. At a high level, the Viper-to-Boogie translation works as follows. Let F and M be the set of Viper fields and methods in the Viper program, respectively. The Viper-to-Boogie translation (1) emits global Boogie declarations G (see Subsection 3.3.6) and (2) generates a separate Boogie procedure $p(m)$ for every Viper method m in M. The intended relation between m and $p(m)$ is given by $\mathsf{Rel}^G_{F,M}(m, p(m))$ in Figure 3.28, which states that the correctness of $p(m)$ w.r.t. G guarantees two things: (C1) the well-formedness of m's specification, and (C2) the correctness of m w.r.t. F and M *if* the specifications of all methods in the Viper program are well-formed. The reason that the correctness of m is not implied *directly* is due to the optimised translation of method calls (as explained in Subsection 3.3.3 on page 127).

Figure 3.28 shows how we generate the certificate of the desired theorem in two steps. First, for each Viper method m and its translated Boogie procedure $p(m)$, we generate a proof for $\mathsf{Rel}^G_{F,M}(m, p(m))$, explained next. Second, we obtain the desired theorem directly from these per-method relational proofs, since the correctness of all Boogie procedures implies that all Viper method specifications are well-formed using (C1), which implies that each Viper method is correct using (C2).

Next, we turn the focus to our strategy for proving $\mathsf{Rel}^G_{F,M}(m, p(m))$. We first discuss the strategy for (C2) (correctness of m), and then discuss the strategy for (C1) (well-formedness of m's specification). Both cases boil down to proving forward simulations, which we establish using our presented methodology.

Intuitively, to prove that m is correct, we have to show that for any state that satisfies m's precondition, executing m's body in this state results in a state that satisfies m's postcondition. The correctness definition for a Viper method (shown in Definition 3.2.1 on page 119) expresses this by requiring that any execution starting in a state σ_v with no permissions that inhales the precondition, then executes the body, and finally exhales the postcondition, cannot fail. As planned, we obtain this result via a forward simulation proof between the executed Viper statement and $p(m)$'s procedure body using our presented methodology. Formally, we show:

$$\exists R', \gamma'.\ \mathsf{stmSim}_{\Gamma_v^0, \Gamma_b^0}(R_0, R', s_v^0, \mathsf{initProgPoint}(p(m)), \gamma')$$
$$\textit{where } s_v^0 \triangleq \texttt{inhale}\ \mathsf{pre}(m); \mathsf{body}(m); \texttt{exhale}\ \mathsf{post}(m)$$

Here, $\Gamma_v^0 \triangleq \mathsf{initCtxt}_v^{M,F}(m)$ is the initial Viper context. Γ_b^0 is a Boogie context that is defined in terms of our chosen type and function interpretation (see Subsection 3.3.6 on page 132). R_0 is a conjunction of an instantiation of the state relation shown in Subsection 3.5.1 on page 155 and a constraint stating that the Viper state has no permissions. $\mathsf{initProgPoint}(p(m))$ is the initial Boogie program point in $p(m)$'s procedure body (as first introduced in Subsection 2.8.3 on page 83). The output state relation R' and output Boogie program point γ' are irrelevant, since we care only about the simulation of failing Viper executions here.

To complete the proof, we choose an initial Boogie state σ_b such that $R_0(\sigma_v, \sigma_b)$. For instance, we pick values for Boogie variables in σ_b that capture σ_v as required by R_0 (*e.g.* the variable `H` modelling the heap would get the Boogie heap value that maps each existing Viper heap location to the same value as the heap in σ_v). As a result, if a Viper execution E_v of statement s_v^0 in σ_v *fails*, the forward simulation provides us with a failing Boogie execution E_b of $p(m)$. Using the correctness of $p(m)$, we conclude that E_b cannot fail, and thus conclude that E_v cannot fail, which concludes the proof of (C2).

There are some steps required to conclude that E_b cannot fail from the correctness of $p(m)$: in particular, we must show that our chosen Boogie type interpretation and function interpretation are well-formed, and that the Boogie axioms are satisfied in σ_b w.r.t. our chosen interpretations. We discussed the former in Subsection 3.3.6 on page 132 and Subsection 3.3.7 on page 136. The latter is fairly straightforward, given our chosen interpretations. Finally, we must bridge the gap between the Boogie AST reduction that reduces each basic command in a separate step (judgement $\rightarrow^*_{\mathsf{AST2}}$ used in the forward simulation) and the Boogie AST reduction that reduces the list of basic commands at the beginning of a statement block in a single step (judgement $\rightarrow^*_{\mathsf{AST}}$ used in the procedure correctness), which is straightforward.[33]

33: Subsection 2.8.2 on page 81 in Chapter 2 discusses the two AST reductions.

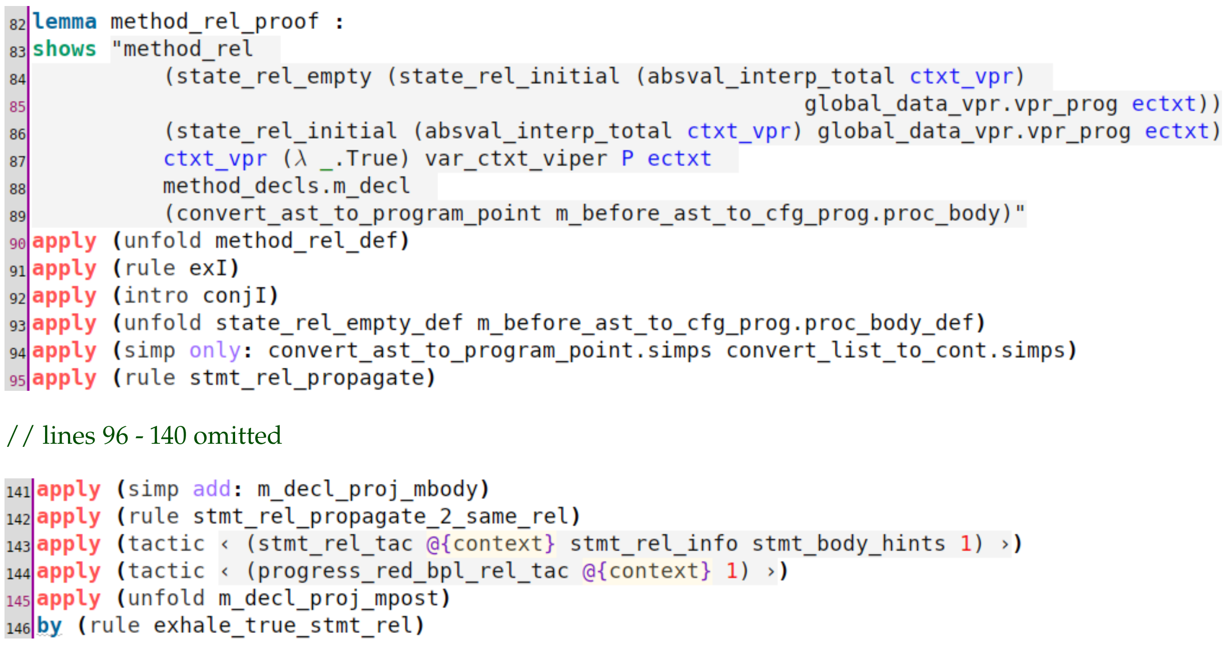
Note that the existing Viper-to-Boogie translation omits well-definedness checks for the final `exhale` of the postcondition in the statement s_v^0 above. This is justified because $p(m)$ checks the well-formedness of m's specification as shown via (C1). To justify the omission of the well-definedness checks, we apply essentially the same strategy as outlined in Subsection 3.5.3 for method calls, where well-definedness checks are omitted for a `remcheck` of the precondition and `inhale` of the postcondition. The main difference is that here in the proof of (C2), we need to explicitly show that the Viper body does not modify m's arguments in order to show that the store of each Viper state in which the final `exhale` of the postcondition is executed is compatible with the precondition. Without such a property, we would not be able to make use of the well-formedness of m's specification, and thus would not be able to justify the existing translation of the final **`exhale`** of the postcondition. In the case of method calls, one gets the fact that the store considered for the postcondition is compatible with the precondition for "free", because the callee's postcondition is considered right after the callee's precondition is exhaled (the store is not modified by an **`exhale`**).

Let us now turn our attention to our strategy for proving (C1) (well-formedness of m's specification). Method m's specification is well-formed iff (1) inhaling the precondition from any well-typed and consistent Viper state σ_v does not fail and (2) inhaling the postcondition from any state whose store is compatible with the precondition does not fail (see Definition 3.3.1 on page 129). We prove the well-formedness of m's specification by showing forward simulations for (1) and (2) using our presented methodology. Formally, we show:

$$\begin{aligned}
&\exists R_1\ \gamma_1.\ \mathsf{stmSim}_{\Gamma_v^0,\Gamma_b^0}(R_0, R_1, \texttt{\textbf{inhale}}\ \mathsf{pre}(m), \mathsf{initProgPoint}(p(m)), \gamma_1) \land \\
&\quad \exists R_2\ \gamma_2.\ \mathsf{sim}_{\Gamma_b^0}(R_1, R_2, \mathit{Succ}, \lambda\_.\ \bot, \gamma_1, \gamma_2) \land \\
&\quad\quad \exists R_3\ \gamma_3.\ \mathsf{stmSim}_{\Gamma_v^0,\Gamma_b^0}(R_2, R_3, \texttt{\textbf{inhale}}\ \mathsf{post}(m), \gamma_2, \gamma_3) \\
&\textit{where } \mathit{Succ} \triangleq \lambda \sigma_v\ \sigma_v'.\ \begin{array}{l}\mathsf{ST}(\sigma_v) = \mathsf{ST}(\sigma_v') \land \mathsf{stateWellTyVpr}(F, m, \sigma_v') \land \\ (\forall l.\ \Pi(\sigma_v')(l) = 0)\end{array}
\end{aligned}$$

In this statement, Γ_v^0, Γ_b^0, R_0, and $\mathsf{initProgPoint}(p(m))$ are identical as in the statement shown for (C1) above. Analogously to the proof for (C2), we can deduce (C1) from this statement by choosing an initial Boogie state σ_b such that $R_0(\sigma_v, \sigma_b)$, and then deducing that (C1) holds, since the are no failing executions from σ_b.

The first conjunct handles the well-formedness requirements on the precondition and the final two conjuncts handle the well-formedness requirements on the postcondition. More concretely, the first conjunct ensures that inhaling the precondition from σ_v cannot fail, because otherwise the corresponding simulation yields a failing Boogie procedure execution from σ_b. The second and third conjuncts handle the postcondition. The second conjunct ensures that if the **`inhale`** of the precondition results in a state σ_v^{pre}, then after the simulation of the **`inhale`** there is a Boogie execution that is able to capture any well-typed Viper state σ_v' that has no permissions and that has the same store as σ_v^{pre} (*i.e.* the store of σ_v' is compatible with the precondition). That is, there is a Boogie procedure execution leading to a Boogie state related to σ_v'. The third conjunct ensures that inhaling the postcondition from such a state σ_v' cannot fail since otherwise the simulation would yield a failing Boogie procedure execution. Since Viper disallows permission introspection in specifications (except in *inhale-exhale assertions*, which we do not support), this implies that inhaling the postcondition cannot fail for any well-typed state with a store that is compatible with the precondition (not just those with empty permissions). This concludes our strategy for proving (C2), and thus our strategy for $\mathsf{Rel}_{F,M}^{G}(m, p(m))$.



```
82 lemma method_rel_proof :
83 shows "method_rel
84         (state_rel_empty (state_rel_initial (absval_interp_total ctxt_vpr)
85                                       global_data_vpr.vpr_prog ectxt))
86         (state_rel_initial (absval_interp_total ctxt_vpr) global_data_vpr.vpr_prog ectxt)
87         ctxt_vpr (λ _.True) var_ctxt_viper P ectxt
88         method_decls.m_decl
89         (convert_ast_to_program_point m_before_ast_to_cfg_prog.proc_body)"
90 apply (unfold method_rel_def)
91 apply (rule exI)
92 apply (intro conjI)
93 apply (unfold state_rel_empty_def m_before_ast_to_cfg_prog.proc_body_def)
94 apply (simp only: convert_ast_to_program_point.simps convert_list_to_cont.simps)
95 apply (rule stmt_rel_propagate)

// lines 96 - 140 omitted

141 apply (simp add: m_decl_proj_mbody)
142 apply (rule stmt_rel_propagate_2_same_rel)
143 apply (tactic ‹ (stmt_rel_tac @{context} stmt_rel_info stmt_body_hints 1) ›)
144 apply (tactic ‹ (progress_red_bpl_rel_tac @{context} 1) ›)
145 apply (unfold m_decl_proj_mpost)
146 by (rule exhale_true_stmt_rel)
```

Figure 3.29: A snippet of an automatically generated Isabelle certificate for a Viper program. The Isabelle lemma in this snippet expresses forward simulations that imply $\mathsf{Rel}^{G}_{F,M}(m, p(m))$ for a concrete method m and concrete Boogie procedure $p(m)$ (see Subsection 3.5.6 for a discussion on $\mathsf{Rel}^{G}_{F,M}(m, p(m))$). Lines 82-89 formally express the statement to be proved (the term `method_rel` is defined in terms of forward simulations) and lines 90-146 form the proof of the lemma (lines 96 - 140 are not shown here). All applied tactics in the proof are built-in Isabelle tactics (such as the `rule` and `simp` tactics) except for the `stmt_rel_tac` and `progress_red_bpl_rel_tac` tactics, which are general custom tactics that we defined and that we use as part of automatically generated certificates. `stmt_rel_tac` (applied on line 143) is our general tactic for automatically proving a forward simulation between a Viper statement and a Boogie statement. The term `stmt_body_hints` given as an argument to `stmt_rel_tac` expresses (automatically generated) hints for the `stmt_rel_tac` tactic for this particular example (see Subsection 3.5.4 for some details of this tactic and corresponding hints).

3.5.7. A Snippet of a Concrete Certificate in Isabelle

To make clearer how our generated Isabelle certificates look at a high level, consider Figure 3.29, which shows a snippet of an Isabelle certificate automatically generated by our tool for a concrete input Viper program and corresponding Boogie program. This snippet shows an Isabelle lemma (and corresponding proof), which expresses forward simulations that imply $\mathsf{Rel}^{G}_{F,M}(m, p(m))$ for a concrete Viper method m in the input Viper program and a concrete Boogie procedure $p(m)$ in the corresponding Boogie program. This entire snippet is automatically generated and Isabelle successfully checks it (and the remainder of the certificate) automatically. In our generated certificate, the proof of $\mathsf{Rel}^{G}_{F,M}(m, p(m))$ uses the lemma shown in the snippet.

This brings our discussion of the key ingredients involved in an application of our general methodology presented in Section 3.4 to an end. All of these ingredients are crucial in order to automatically generate certificates for concrete front-end translations. We will now continue with the non-technical part of the chapter, starting with the evaluation of our certificate-producing instrumentation of the existing Viper-to-Boogie translation.

3.6. Implementation and Evaluation

We instrumented the existing Viper-to-Boogie verifier implementation such that on every run the implementation automatically produces an Isabelle certificate justifying the soundness of its translation to Boogie. We evaluated this instrumented version on a diverse set of Viper benchmarks, to check that our automation actually enables automatically-checkable certificates in practice.

3.6.1. Implementation

Given an input Viper program, the Viper-to-Boogie implementation passes the generated Boogie program to the Boogie verifier as a text file, and our instrumented version of the implementation automatically generates a soundness certificate for this particular verifier run. Our approach for generating the soundness certificate connects the input Viper AST as represented by the Viper-to-Boogie implementation *directly* to the target Boogie AST *as represented by Boogie verifier*. The alternative would have been to connect the input Viper AST to the target Boogie AST representation used by the Viper-to-Boogie implementation itself. Our choice has multiple advantages. First, we do not have to trust the Boogie parser that parses the Boogie text file into the internal Boogie AST representation. Second, our work in Chapter 2 already provides infrastructure for working with the internal Boogie AST representation (including generating the Isabelle embedding of the Boogie AST representation). As a result, we need not reimplement any infrastructure that deals with any Boogie AST representation as part of our certificate-producing instrumentation of the Viper-to-Boogie implementation, which saves some work. Third, our approach generalises to verifier implementations that directly target the Boogie verifier's AST such as Dafny [3] (instead of passing the Boogie file as a text file to Boogie). We discuss an advantage of the alternative (connecting to the Boogie AST representation used by the Viper-to-Boogie implementation) as part of future work in Subsection 3.9.4.

[3]: Leino (2010), *Dafny: An Automatic Program Verifier for Functional Correctness*

We make the following four adjustments to the Viper-to-Boogie implementation. First, we desugar its usages of polymorphic maps as described in Subsection 3.3.6 on page 132, since there is no formal model for polymorphic maps. Second, we adjust the implementation to not emit Boogie declarations or commands that are used only for features outside of our subset (the original implementation always emitted those without checking whether the corresponding features were actually used). Third, we switch off simple syntactic transformations that the Viper-to-Boogie implementation applies to the produced Boogie program (*e.g.* constant folding, elimination of if-statements with no branches), since we do not support them yet. Justifying those syntactic transformations should be straightforward and is orthogonal to our work, because none of the three main challenges that we discuss in Section 3.1 for front-end translations arise for these transformations. Fourth, we introduce a `havoc` statement in the Boogie program at the point when a scoped Viper variable is introduced, which faithfully models the semantics of such a variable. The original Viper-to-Boogie implementation instead just introduces a fresh Boogie variable at the beginning of the Boogie procedure. The reason for this fourth change is because forward simulations cannot capture the

Table 3.1: Overview of benchmarks and results. For each test suite, we report the number of Viper files, the total number of Viper methods contained in those files, as well as the *mean* number of non-empty lines of code for the Viper files, Boogie files, and produced Isabelle certificates. We measured the mean and median time it took to check the Isabelle certificates in seconds.

Test suite	Files	Methods	Viper	Boogie	Isabelle	Cert. Check	
	no.	no.	Mean [LoC]	Mean [LoC]	Mean [LoC]	Mean [s]	Median [s]
Viper	34	105	33	298	1719	33.8	23.8
Gobra	17	65	60	287	1937	32.7	25.3
VerCors	18	116	63	332	2930	43.1	40.9
MPP	3	13	206	1060	5164	109.0	46.2
Overall	**72**	**299**	**54**	**335**	**2217**	**39.0**	**32.9**

translation otherwise. However, using a different simulation, proving the equivalence of both translations is straightforward.[34]

34: We discuss the generalisation of our work to other simulations as part of future work in Subsection 3.9.2.

3.6.2. Evaluation

Our evaluation answers the questions: **(RQ1)** Does our implementation generate certificates that Isabelle can check successfully (and automatically) for a diverse set of examples? **(RQ2)** Does Isabelle check the generated certificates in reasonable time (*e.g.* is the check feasible as part of continuous integration)?

To obtain a diverse set of representative examples, we considered the Viper test suite as well as the test suites of three tools that produce Viper code: Gobra [5] (for Go), VerCors [6] (for Java), and MPP [108] (a tool performing a modular product transformation on Viper programs). To eliminate trivial translations, we focused on Viper programs that use the heap, as indicated by the occurrence of at least one accessibility predicate. Out of those, we included all Viper programs that fall into our supported Viper subset. We followed different strategies to systematically obtain additional examples from the different test suites. For Viper and MPP, we additionally considered files with three kinds of features that we do not support: *old*-expressions, *new*-statements, and method calls whose arguments are not variables (we currently support only variables as method call arguments). We were able to bring these files into our support subset as follows. We manually removed those parts that contain old expressions (for instance, leading to the verification of weaker postconditions, since old expressions often appear in postconditions). We manually desugared *new*-statements into our subset (using scoped variables and `inhale` statements). Finally, we manually made sure that each argument to a method call is a variable (e.g. we rewrote `m(i+1)` to `var t := i+1; m(t)`). For Gobra and VerCors, we removed boilerplate code that is emitted for each file if the boilerplate code was not required for the main example code (otherwise, we did not include the file). After this removal and resulting selection, we followed the same process as for Viper and MPP for old expressions, new-statements, and method calls whose arguments are not variables. Moreover, we included files generated by Gobra that had at most two occurrences of features outside of our subset if those could be manually desugared into our subset (*e.g.* eliminating a function by inlining its body).

[5]: Wolf et al. (2021), *Gobra: Modular Specification and Verification of Go Programs*

[6]: Blom et al. (2017), *The VerCors Tool Set: Verification of Parallel and Concurrent Software*

[108]: Eilers et al. (2018), *Modular Product Programs*

Table 3.2: Detailed results of our evaluation for a selection of files showing the number of methods, the nonempty lines of code for the Viper program, Boogie program, and produced Isabelle certificate, and the time it took to check the proof in seconds.

Test suite	**File**	**Methods**	**Viper**	**Boogie**	**Isabelle**	**Cert. Check**
		no.	Total [LoC]	Total [LoC]	Total [LoC]	Total [s]
Viper	testHistoryProcesses	13	205	1711	7035	126.3
Gobra	defer-simple-02	9	211	853	4717	60.6
VerCors	inv-test-fail2	5	92	514	2596	56.5
MPP	banerjee	8	414	2014	9545	242.4
MPP	darvas	2	91	582	2800	38.4
MPP	kusters	3	112	583	3146	46.2

As summarised in Table 3.1, through the above process we collected a total of 72 Viper files (containing 299 methods), with a mean of 54 non-empty lines of code per file. Among these 72 files, the file with the most non-empty lines has 414 non-empty lines of code. We ran our implementation on all 72 Viper files to generate the Boogie translations and the Isabelle certificates, and measured the time it took for Isabelle to check the generated certificates (the mean of five repetitions). The measurements were run on a ThinkPad X1 Yoga Gen 5 on Ubuntu 20.04 with 16 GB RAM and i7-10510U @ 1.8 GHz (scaled up to 4.9 GHz using Turbo Boost). The generated Boogie translations are on average 6.2x larger (335 non-empty LoC on average), partially illustrating the semantic gap between Viper and Boogie.

Isabelle successfully checked the generated certificates for all Viper files, including the Viper programs automatically generated by other tools. This shows that our approach is effective for practical verifiers and answers RQ1 positively. The resulting Isabelle certificates have on average over 2000 lines and are checked on average in less than a minute.

Out of the 72 files, the Viper-to-Boogie implementation successfully verifies 26 files, and reports verification errors for the remaining 46 files. This is expected since files in verifier test suites often contain expected errors in some of the methods to test whether the verifier catches them. Our generated certificates are useful in both cases (verification success and verification failure). In case of verification success, our certificates formally prove that the input Viper program is correct if the corresponding Boogie program is correct (which is the case if the Boogie verifier is sound). In case of verification failure, our certificates formally prove that if the input Viper program is indeed incorrect (which need not be the case due to incompletenesses), then the corresponding Boogie program is incorrect. As a result, in this second case, our certificates guarantee that even if we use a possibly more complete Boogie verifier implementation compared to the actual one (*e.g.* using a more complete SMT solver), then the Viper-to-Boogie implementation will not report verification success for an incorrect Viper program (as long as this more complete Boogie verifier implementation is sound). Finally, note that our generation and checking of certificates is completely independent from whether the Viper-to-Boogie verifier implementation reports success or failure (or whether a Viper program is correct or not).

The detailed results for each evaluated example in each of the test suites are shown in the appendix (App. A.1). Table 3.2 shows the detailed results for some of the more complex examples: Table 3.2 contains the largest example (in terms of lines of Viper code) from each of the test suites, and contains the two remaining examples from MPP. These examples are complex w.r.t. those considered in the evaluation due to their size and due to the fact that the MPP examples are drawn from different research papers. They show that our tool can certify challenging programs.

For this selection, the times to check the certificates range from 38 seconds to 4 minutes. No file in any of the 72 files takes longer than 4 minutes to check. These times are acceptable, since we expect the validation to be performed occasionally (in particular, before the verified program is released or as part of continuous integration), but not on every run of the verifier. Thus, we answer RQ2 positively for the considered 72 files. To obtain additional representative files from the test suites, we would need to extend our supported Viper subset.

Most parts of our certificates are not yet optimised to make certificate checking faster. For example, variable accesses currently result in an overhead in the certificate that is linear in the number of active variables, respectively. As another example, we generate a lemma and a corresponding proof for each declared field that states what its declared type is, and then use these lemmas for different proofs in the certificate. The time complexity for Isabelle to check such a particular lemma is linear in the number of declared fields, since the declared fields are represented via a list, and the proof needs to essentially look up the field in this list by linearly going through the list. Both of these examples (variable accesses and field proofs) could be improved by having more efficient ways of representing data in the certificate. For instance, one could represent variables (resp. fields) using a binary search tree, which would make looking up a specific variable (resp. field) only logarithmic in the number variables (resp. fields). The seL4 kernel verification code has Isabelle infrastructure for using such a binary search tree representation in proofs [98].

[98]: seL4 Developers (n.d.), *Efficient lookup table creation in Isabelle*

3.6.3. Trusted Components

Our certificate-producing version of the Viper-to-Boogie implementation greatly reduces the parts of the verifier implementation that must be trusted. In particular, if Isabelle successfully checks the generated certificate, then the code translating a Viper program to the corresponding Boogie program need not be trusted.

However, there are still various components that must be trusted in order to conclude that a certificate successfully checked by Isabelle actually implies that the correctness of the Boogie program produced by the Viper-to-Boogie implementation implies the correctness of the input Viper program. These trusted components include:

- the soundness of the Viper parser that translates a source program represented in text into Viper's internal AST representation
- our deep embedding of the Viper AST representation in Isabelle (including its semantics via our Viper language formalisation) must reflect the input Viper program

- our deep embedding of the Boogie AST representation in Isabelle (including its semantics via our Boogie language formalisation) must reflect the Boogie program
- the soundness of Isabelle

In Subsection 2.10.3 on page 88, we already discussed how one could increase the trustworthiness of parsers and our deep embedding of Boogie AST representations, and we also emphasised the trustworthiness of Isabelle. The semantics of the embedded Viper program is a fundamental trust assumption, which we cannot fully eliminate, since soundness is defined w.r.t. this semantics. To increase the confidence in this semantics, we can take similar measures as discussed for the Boogie semantics in Subsection 2.10.3.

To conclude that a Viper program is correct for a successful verification result under the assumption that the verification condition generated by the Boogie verifier is valid, one could combine our generated certificate for the Viper-to-Boogie implementation with our generated certificate for the Boogie verifier implementation (the latter was presented in Chapter 2). As we discuss in Subsection 3.9.5 on page 201, our work in Chapter 2 must be extended to achieve this, but such an extension should not require fundamental insights.

Viper's type checker

The role of Viper's type checker is analogous to the role of Boogie's type checker for certificates generated in Chapter 2 (see our discussion on the Boogie type checker in Subsection 2.10.3). We need not trust Viper's type checker, since our certificates do not explicitly assume that the input program is well-typed. However, since our semantics accurately models only Viper programs that are well-typed, we could weaken our trust assumption on the Viper semantics by proving a type soundness result for our operational semantics. This would increase our confidence in our Viper semantics if the input program is well-typed, which we know is the case if we trust a successful result by Viper's type checker or if we automatically prove well-typedness.

Combining certificates with Viper as an IVL

Viper is mainly used as an IVL for some front-end. Most of these front-ends directly construct a Viper AST as used by the Viper-to-Boogie implementation without using any parser. As a result, the soundness of the Viper parser is not relevant for most Viper front-ends. Even if a Viper front-end uses a Viper parser, one need not trust the parser, if one directly shows that the correctness of the front-end program is implied by the correctness of the corresponding Viper encoding as represented by the Viper AST representation used in the Viper-to-Boogie implementation. Moreover, if one then connects such a result between a front-end program and the Viper AST representation with the certificate generated by our certificate-producing Viper-to-Boogie implementation, then one also need not trust the Isabelle embedding of the Viper AST representation and the Viper semantics. However, one would have to trust the Isabelle embedding of the Boogie AST representation and the Boogie semantics.

3.7. Related Work

Front-end soundness

[27]: Lehner et al. (2007), *Formal Translation of Bytecode into BoogiePL*

[28]: Vogels et al. (2009), *A Machine Checked Soundness Proof for an Intermediate Verification Language*

[30]: Gössi (2016), *A Formal Semantics for Viper*

[109]: Leino et al. (2009), *A Basis for Verifying Multi-threaded Programs*

[110]: Leino et al. (2009), *Verification of Concurrent Programs with Chalice*

[29]: Backes et al. (2011), *Automatically Verifying Typing Constraints for a Data Processing Language*

[45]: Dardinier et al. (2025), *Formal Foundations for Translational Separation Logic Verifiers*

Various works prove the soundness of front-end translations *once and for all*. For instance, Lehner and Müller [27] prove a translation from Java Bytecode to Boogie, Vogels et al. [28] target a translation from a toy object-oriented programming language to Boogie, and Gössi [30] targets a translation from Chalice [109, 110] to Viper. These proofs are done on paper and do not consider an actual implementation of the translation. Backes et al. [29] prove a translation sound from the Dminor data processing language to the Bemol IVL in Coq. They do not provide a proof connecting the formalised translation to their F# implementation. In work not presented in this dissertation, we prove a translation from a simple concurrent language to Viper sound in Isabelle [45]. This translation does not reflect an actual implementation used in practice; thus many of the challenges that we tackle in this chapter do not show up there. However, this translation encodes a front-end *program logic* into the IVL, which is common for translations into Viper. The corresponding soundness proof of this translation connects a front-end program logic with an *axiomatic semantics* of the IVL. In contrast, this chapter's approach connects a front-end operational semantics with an IVL operational semantics, which fits better for translations such as the Viper-to-Boogie translation that essentially encode the operational semantics, but is less convenient for translations that encode a program logic into the IVL; we discuss this as part of future work (Subsection 3.9.3 on page 200). Summers and Müller [111] and Wolf et al. [112] present more intricate translations used by implementations that also encode front-end program logics into Viper. They reason about front-end soundness via proof sketches on paper, which explore only the high-level reasoning principles and thus avoid many of the complexities involved in a fully formal proof. Herms [25] proves a translation from C to the WhyCert IVL (inspired by the Why3 IVL) sound in Coq, which they then turn into an executable tool via Coq's extraction to OCaml. The resulting tool has similarities to the Jessie Frama-C implementation [113], which translates C programs to Why3; Herms [25] discusses mismatches between their mechanisation and the Jessie implementation. In contrast, as we have shown, our certification methodology can be applied to existing front-end implementations, which are typically implemented in efficient mainstream programming languages, use diverse libraries, and include subtle optimisations omitted from idealised implementations.

[111]: Summers et al. (2020), *Automating deductive verification for weak-memory programs (extended version)*

[112]: Wolf et al. (2022), *Concise outlines for a complex logic: a proof outline checker for TaDA*

[25]: Herms (2013), *Certification of a Tool Chain for Deductive Program Verification*

[113]: Marché et al. (2018), *The Jessie plugin for Deductive Verification in Frama-C*

Internal program transformations

As discussed in Chapter 1, translational verifiers perform a series of program transformations, *e.g.* by translating programs to a lower-level IVL or *internally* without changing the language (e.g. monomorphisation, or transformations such as those presented in Chapter 2). Our approach in this chapter can in principle be applied to both kinds of transformations, but is tailored towards the former, where the semantic gap is large, non-local checks arise, and diverse translations are used. Existing work for the validation of internal transformations does not provide solutions for these challenges. For instance, our work in Chapter 2 validates the verification condition (VC) generation implementation of Boogie programs, which includes some internal Boogie-to-Boogie transformations. In these transformations, the semantic gap is small (the source and target constructs are largely the same), and thus the decomposition into smaller problems is immediate, while in this chapter the decomposition is a challenge. Moreover, Chapter 2 need not deal with non-local checks and diverse translations.

The work in Chapter 2 tackles different challenges for the internal transformations that requires the use of different kinds of simulations. For instance, the certification of assignment elimination tracks a set of Boogie states for a single Viper state, which cannot be captured by the kind of forward simulations we use in this chapter; it would be interesting future work to apply this chapter's methodology to other kinds of simulations such as the one used for assignment elimination. Besides internal transformations, Chapter 2 connects a Boogie program and the corresponding VC; this chapter considers only program-to-program transformations.

Another difference is that for the cycle elimination and assignment elimination transformations in Chapter 2 it seems that per-run validation is significantly easier than proving the transformations once and for all. The reason is that various nontrivial computations performed by Boogie and properties ensured by Boogie for these transformations need not be explicitly reflected in the generated certificates. These are instances where a nontrivial computation produces a result that can be validated without knowing how the computation was performed, and where a property can be implicitly validated and used without needing to explicitly state the property. An example of the former is Boogie's nontrivial computation of back edges. The generated certificate does not need to know how Boogie eliminates back edges, but just needs to ensure that the back edges that were computed are the correct ones. An example of the latter is the property that the source and target CFGs of Boogie's assignment elimination must be acyclic. This property is never explicitly stated, but the generated certificate implicitly makes sure that the source and target CFGs are indeed acyclic (otherwise, the certificate is not valid and Isabelle would thus fail to check it). For the Viper-to-Boogie implementation applied to our considered Viper subset there were no such instances. Nevertheless, as we discuss in Chapter 1, per-run validation still has a major advantage over once-and-for-all proofs, since it is easier to provide formal guarantees for *existing* implementations, as we do for the Boogie and Viper verifier implementations, and is a general goal of this dissertation.

Finally, the work in Chapter 2 can in principle be combined with work in this chapter to enable end-to-end soundness guarantees for Viper, but this would require extending the Boogie verifier validation to an additional internal Boogie transformation (*i.e.* the dead variable elimination) and to a larger Boogie subset (see Subsection 3.9.5 on page 201 for more details).

Forward simulation automation

[114]: Rizkallah et al. (2016), *A Framework for the Automatic Formal Verification of Refinement from Cogent to C.*

Multiple works also embed programs in an interactive theorem prover (ITP) and then automate forward simulation proofs. Rizkallah et al. [114] define a refinement calculus for the Cogent compiler to automatically prove a forward simulation in Isabelle for a Cogent expression and its C translation. Their calculus includes syntax-directed rules for deriving simulation judgements, but these rules do not provide the abstraction we needed to handle diverse translations. For instance, their calculus does not provide the abstraction needed to use the same set of rules for justifying different translations of the same construct that are possibly justified by non-local checks. Our rules achieve this abstraction by allowing changes to the state relation as part of a simulation proof; additionally, we provide a systematic way of making these adjustments. The Cogent compiler was developed with validation in mind: for instance, the compiler applies transformations to the input Cogent program before translating a Cogent representation to a C representation (which is the translation handled by the refinement framework), making sure that this translation step to C is kept simple, which simplifies validation. In contrast, our goal was to validate existing verifier implementations with all their intricacies. It would be an interesting research direction to investigate how to develop verifier implementations such that their validation is simpler.

[115]: Winwood et al. (2009), *Mind the Gap*
[116]: Cock et al. (2008), *Secure Microkernels, State Monads and Scalable Refinement*
[117]: Klein et al. (2010), *Refinement in the Formal Verification of the seL4 Microkernel*

The verification of the seL4 kernel includes two large forward simulation proofs in Isabelle, for which proof automation was developed [115–117]. This automation reduces the manual proof overhead, but still requires user interaction. In contrast, our validation certificates are generated and checked completely automatically. Like us, the authors of those works prove rules to decompose the simulation for composite statements syntactically but, contrary to us, do not decompose statements semantically into smaller simulations. They use standard Hoare triples, for which they have separate automation, for different purposes. For instance, they reduce the proof of certain simulation judgements to proofs of Hoare triples, and they use Hoare triples to express properties on the source program (resp. target program) in premises of some of their rules.

Compilers

[118]: Pnueli et al. (1998), *Translation Validation*

[37]: Tristan et al. (2008), *Formal verification of translation validators: a case study on instruction scheduling optimizations*
[38]: Tristan et al. (2009), *Verified validation of lazy code motion*
[41]: Gourdin et al. (2023), *Formally Verifying Optimizations with Block Simulations*

[40]: Kang et al. (2018), *Crellvm: verified credible compilation for LLVM*

[119]: Rinard et al. (1999), *Credible Compilation with Pointers*

Translation validation [118] approaches for compilers define a per-run validator, which checks whether the compilation is sound. *Formal* translation validation additionally proves the correctness of the validator formally using an interactive theorem prover (ITP) w.r.t. a formalised semantics for the source and target program. One common formal approach is to express and prove the correctness of the validator *once and for all* in the ITP (instead of generating a certificate in the ITP), and then to extract executable code for the validator (the extraction must typically be trusted). This approach is used, for example, for some compilation passes in the CompCert C compiler project [37, 38, 41] and also for LLVM optimisations [40], both of which are formally proved in Coq. One aspect that distinguishes the validator developed for LLVM optimisations [40] is that the validator uses a *credible compilation* approach [119], where a general relational program logic, which is not tied to a specific translation, is proved sound, and the validator tries constructing a proof in this relational logic.

For many validators that are formally justified, the source and target languages are similar. In contrast, Sewell et al. [120] use a combination of different approaches (*e.g.* using two different ITPs and an SMT solver) to define a validator for a translation where the source and target language are very different: they validate the GCC compiler translation from C to binary code. In particular, the final part of the validator relies on a refinement proof using an SMT solver.

[120]: Sewell et al. (2013), *Translation validation for a verified OS kernel*

It would be interesting to explore the feasibility of extracting a validator from an ITP to executable code for front-end translations, where the semantic gap between the languages is large. One difference to compiler transformations is that front-end translations incorporate reasoning steps, such as assumptions and proof obligations prescribed by a program logic. This encoding is achieved via components not present in executable languages such as assume statements, havoc statements, and background axiomatisations. Moreover, front-end translations emit code that checks nontrivial properties that are then relied upon in other parts of the encoding.

Validation of verifier implementations

Validation has also been used to obtain formal guarantees for implementations of verifiers, but none of the existing works target front-end translations and the challenges they entail. Lin et al. [48] and Wils and Jacobs [46] validate verifiers obtained via the K framework (by generating Metamath certificates on every run) and VeriFast (by generating Coq certificates on every run), respectively. These verifiers use symbolic execution, which works very differently compared to translations in translational program verifiers. As a result, the applied validation approaches differ fundamentally from our validation approach. Garchery [35] validates certain logical transformations in Why3. Cohen and Johnson-Freyd [36] also prove such logical transformations, but do so once and for all in Coq to demonstrate their Why3 mechanisation. Neither of the two consider the actual verification condition generation.

[48]: Lin et al. (2023), *Generating Proof Certificates for a Language-Agnostic Deductive Program Verifier*

[46]: Wils et al. (2023), *Certifying C program correctness with respect to CH2O with VeriFast*

[35]: Garchery (2021), *A Framework for Proof-carrying Logical Transformations*

[36]: Cohen et al. (2024), *A Formalization of Core Why3 in Coq*

Formal results for Viper

In work not presented in this dissertation that explores a complementary research direction, we formalise the semantics for a generic IVL in Isabelle, which includes `inhale` and `exhale` statements, and which has various parameters such as the state model [45]. This semantics was designed to be independent from back-end verifiers for Viper (such as the Viper-to-Boogie implementation) and suitable to reason about the soundness of front-end translations into Viper. As part of the Isabelle mechanisation, the parameters of this generic IVL are instantiated to obtain a subset of Viper with an accompanying Viper semantics (that we call *ViperCore* here), which comes in two flavours: an operational semantics that is suitable for the connection to a Viper back-end and

[45]: Dardinier et al. (2025), *Formal Foundations for Translational Separation Logic Verifiers*

an equivalent axiomatic semantics that is suitable for reasoning about front-end translations into Viper.

The ViperCore semantics abstracts away Viper back-end verifier details. This is in contrast to the semantics formalised in this chapter, which exposes some details of the Viper-to-Boogie translation in order to simplify the validation as discussed in Subsection 3.2.8 on page 122. One detail exposed in the semantics formalised in this chapter is that nondeterministic heap assignments are performed as part of an `exhale` instead of an `inhale`. This is slightly unintuitive, because the property one needs is that whenever an `inhale` operation obtains permission to a heap location for which there was no permission before, then the semantics must consider every possible heap value for this location. ViperCore instead performs nondeterministic heap assignments at an `inhale`, which is more intuitive. Moreover, ViperCore uses a partial heap model instead of a total heap model used in this chapter's semantics; as a result, when exhaling all the permission to a heap location, the corresponding location is removed from the domain of the partial heap. There is a formal soundness proof connecting the two semantics [45], which shows that correctness w.r.t. the semantics formalised in this chapter implies correctness w.r.t. ViperCore. This result provides further confirmation that this chapter's semantics is a valid one, and also provides further confirmation that ViperCore's semantics is a valid one (since this result combined with our certificates shows that ViperCore captures the existing Viper-to-Boogie translation).

[45]: Dardinier et al. (2025), *Formal Foundations for Translational Separation Logic Verifiers*

Another difference between ViperCore and this chapter's semantics is the following: in ViperCore, `inhale` and `exhale` are expressed with *semantic* assertions as input (specifying when an assertion is *satisfied*) instead of the syntactic assertions used in our case. That means `inhale` and `exhale` are not specified themselves operationally as in this chapter's semantics. This chapter's semantics operationally traverses the syntactic input assertion from left to right, which accurately captures the meaning of permission introspection as part of `inhale` and `exhale` (permission introspection is currently not compatible with ViperCore). As a result, ViperCore's semantics can be used with any assertion as long as one can express the satisfaction of that assertion semantically (*i.e.* as a function from states to Booleans).

To use ViperCore for concrete Viper programs, an interpretation function is provided that transforms *syntactic* Viper expressions and assertions into their semantic counterparts. Currently, this function supports the entire subset formalised in this chapter except for permission introspection. Providing an interpretation such that `inhale` and `exhale` are modelled accurately when assertions have permission introspection is not directly possible in the current setup. Additionally, the interpretation function supports *wildcard* permission amounts in accessibility predicates (*i.e.* existentially quantified permissions). The interpretation function for the separating conjunction states that a state σ_v satisfies A && B if σ_v can be split into two states (in terms of the permissions) such that A is satisfied in one state and B is satisfied in the other. Such a definition more directly reflects the original separating conjunction definition from separation logic. Moreover, it ensures that the separating conjunction is commutative, which is not the case in the semantics defined in this chapter. For instance, `inhale x.f == 2 && acc(x.f, 1/2)` fails in the semantics defined in this chapter in a state without any permissions, while it would not fail in the ViperCore **inhale**. However, the Viper verifier cannot verify this **inhale** since the verifier decomposes the assertion from left to right, so our semantics is accurate enough. Moreover, note that the ViperCore interpretation for separating conjunctions does not capture permission introspection, since permission introspection requires a left-to-right decomposition.

ViperCore includes a subset of Viper statements formalised in this chapter and additionally includes a *havoc* command (analogous to the Boogie **havoc** command), but the statements that ViperCore does not support (but which this chapter does) are expressible via the others. In particular, ViperCore does not support method calls and **assert** statements, both of which are expressible via **inhale** and **exhale** (and havoc for calls that return results).

Gössi [30] provided the first formal operational Viper semantics on paper, which is at a similar abstraction layer as the formal Viper semantics defined in this chapter[35] and targets a larger subset than us (*e.g.* including predicates, functions, and wands). They prove the soundness of a translation from Chalice [109, 110] to Viper w.r.t. their semantics. Their work is a good first attempt at formalising the Viper semantics, which helped gain a better understanding for Viper. We have since learnt various lessons that led to a different semantics and to different ways of formalising Viper features.

[30]: Gössi (2016), *A Formal Semantics for Viper*

35: For instance, their semantics also uses a total heap and nondeterministically chooses values for heap locations as part of **exhale** but not **inhale**.

[109]: Leino et al. (2009), *A Basis for Verifying Multi-threaded Programs*

[110]: Leino et al. (2009), *Verification of Concurrent Programs with Chalice*

In the following, we discuss one example where we took a different decision and one example for a feature outside of our formalised subset where we would take a different decision. In their semantics, **exhale** A && B is defined as the sequential composition of **exhale** A and **exhale** B. That is, there is not a separate operation (such as **remcheck** A && B in our case) that first removes all the permissions without changing the heap and only then chooses values for heap locations nondeterministically. Such a semantics does not accurately capture Viper, since it, for example, makes **exhale** **acc**(`x.f`) && `x.f` > 0 always fail, as `x.f` would always be evaluated in a state without permission to `x.f`. A second example is the treatment of Viper predicates (a feature outside of our formalised subset). As we discovered and will discuss in Subsection 3.9.1 on page 195, to accurately capture predicates, a Viper semantics must extend the state consistency notion (or must do something similar). Otherwise, the resulting semantics cannot, for instance, justify the existing Viper-to-Boogie translation. Their semantics does not take this extended consistency notion into account. This latter point also motivates a fairly different formalisation of Viper predicates in general, which we are currently exploring. The high-level idea is to add more information on predicates to the state model: in particular, reflecting the recursive structure of the predicates directly in the state.

Viper supports two back-end verifiers: the Viper-to-Boogie implementation, which this chapter makes certifying for a Viper subset, and a symbolic execution back-end. There are various formalisations of the symbolic execution back-end. Schwerhoff [121] formalises the core parts of the symbolic execution back-end for a substantial Viper subset on paper (*e.g.* predicates and iterated separating conjunctions in addition to our formalised subset). In work not presented in this dissertation, we

[121]: Schwerhoff (2016), *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*

[45]: Dardinier et al. (2025), *Formal Foundations for Translational Separation Logic Verifiers*

mechanise a subset of the symbolic execution back-end in Isabelle [45] in which certain implementation details are abstracted away compared to the model formalised by Schwerhoff [121]. Moreover, we formally prove this back-end is sound w.r.t. the *ViperCore semantics*. Zimmerman et al. [44] provide an alternative formalisation of the symbolic execution back-end for a subset of Viper on paper, but they extend the formalisation to additionally deal with *gradual* specifications. They use this formalisation to prove soundness of the gradual verifier Gradual C0 [122], which uses the Viper symbolic execution back-end extended with support for gradual specifications.

[44]: Zimmerman et al. (2024), *Sound Gradual Verification with Symbolic Execution*

[122]: DiVincenzo et al. (2022), *Gradual C0: Symbolic Execution for Efficient Gradual Verification*

Finally, there are further works that provide a formal justification for a variety of aspects related to Viper. Dardinier et al. [123] provide a formal model for *fractional* predicates as used in Viper (and VeriFast [89]). In work not presented in this dissertation, we prove a novel automation approach for magic wands (a separation logic connective) sound [65], which we then apply to Viper, and we prove that (bounded) method call inlining and loop unrolling is verification preserving in automated SL verifiers (including Viper) under certain conditions [64].

[123]: Dardinier et al. (2022), *Fractional resources in unbounded separation logic*

[89]: Jacobs et al. (2011), *VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java*

[65]: Dardinier et al. (2022), *Sound Automation of Magic Wands*

[64]: Dardinier et al. (2023), *Verification-Preserving Inlining in Automatic Separation Logic Verifiers*

Other related work

Boogie developers have added an option to monomorphise polymorphic maps in Boogie programs via non-polymorphic maps [99]. This option provides an alternative to ours for desugaring polymorphic maps, which, in the case of Viper, circumvents the circularity challenge discussed in Subsection 3.3.6, since Viper does not permit storing heaps in fields. However, in general, front-ends may permit storing heaps in fields.

[99]: Qadeer (2022), *Monomorphization of polymorphic maps and binders*

Smans et al. [33] prove soundness once and for all on paper of a verification condition generator for a language with implicit dynamic frames (IDF) assertions that does not use an IVL. They also implement a prototype, but do not formally connect the proof to the implementation. We also applied our methodology to a verifier based on IDF, but validate an actual implementation.

[33]: Smans et al. (2012), *Implicit Dynamic Frames*

3.8. Impact of Work on Viper

The work in this chapter, which led to a certificate-producing Viper-to-Boogie verifier implementation, had a positive impact on the Viper ecosystem. A large part of this positive impact was a result of needing to deeply understand the Viper language and the Viper-to-Boogie implementation in order to produce certificates. Our positive impact can be categorised into four dimensions. First, we improved the Boogie encoding generated by the verifier such that the encoding is easier to understand for developers of the verifier and also such that errors reported by the verifier are more intuitive for Viper users. Second, we improved the Viper-to-Boogie implementation (*e.g.* by removing functionality that was duplicated or by simplifying code). Third, we identified two soundness bugs. One of these was within our formalised Viper subset, which we fixed [124]. The other one was a subtle bug outside of our formalised subset, which we documented but which has not yet been fixed because

[124]: Parthasarathy (2023), *Include definedness checks during exhale (Pull Request 457)*

fixing the bug requires answering design questions for Viper [125]. This bug arises in a corner case that likely does not impact practical Viper programs, but fixing the bug is still important and will improve Viper as a language due to the design questions that arose as a result of this bug. We will discuss both of these soundness bugs in more detail later in this section. Fourth, we noticed that the semantics of a feature outside of our formalised Viper subset (*unfolding expressions*) is unclear in combination with permission introspection [126]. This lack of clarity does not impact practical Viper programs, but resolving the meaning of this combination is important for clarifying the meaning of both features and also will help future use cases that may rely on this combination.

[125]: Parthasarathy (2024), *Viper allows predicates whose fractional amount is not self-framing (Issue 809)*

[126]: Parthasarathy (2023), *The semantics of permission introspection in the body of unfolding expressions is unclear (Issue 682)*

It is particularly interesting that our work had impact *beyond* the formalised subset. This is partially because we always tried to keep Viper as a whole in mind even though we finally focused on a core subset, because our goal is to eventually support the entire subset. However, another reason is that the work on our core subset made certain questions explicit for which we had to find answers formally, and as a result it was natural to ask the same questions for features outside of our subset.

In this section, we highlight some of the concrete instances where our work had a positive impact. First, we discuss the encoding of well-definedness checks performed as part of an `exhale` and `inhale`. The improvements affected the first three dimensions (encoding, implementation, soundness). Second, we discuss two encoding improvements, one of which we implemented and another for which our work would be important to ensure soundness. Third, we present the discovered soundness bug outside of our formalised subset. We do not discuss the fourth dimension in this section (unclear semantics of a Viper feature). Instead, we refer interested readers to the corresponding GitHub issue [126], which explains why the semantics of permission introspection is unclear in the presence of *unfolding expressions*.

3.8.1. Well-Definedness Checks for `exhale` and `inhale`

`exhale` results in failure if an expression evaluated during the corresponding reduction is ill-defined. As a result, the Viper-to-Boogie implementation emits well-definedness checks in the Boogie encoding for each of these expressions except in optimised cases such as the `exhale` of a method call precondition. Before our work, the implementation encoded the well-definedness of expressions as follows for `exhale` A. In a first step, the implementation emitted well-definedness checks for all expressions in A without performing the removal of the permissions, and then, in a second step, the implementation encoded the removal of the permissions without performing the well-definedness checks. This does not directly correspond to the semantics formalised in this chapter, which checks whether expressions are well-defined on the fly (*i.e.* as part of the evaluation of each expression). As a result, this prior encoding has two downsides: (1) if `exhale` A fails, then in certain cases the user is presented with unintuitive error messages, and (2) the encoding can be unsound in the presence of permission introspection.

The following example illustrates the first downside (unintuitive error messages):

```
exhale acc(x.f) && 0/0 == 0/0
```

Suppose this `exhale` is executed in a state without any permissions. Then, this `exhale` fails because there is insufficient permission for the first conjunct. However, the error reported using the prior encoding was that `0/0` is ill-defined due to the division by 0. While it is true that the assertion syntactically contains a division by 0 and division by 0 leads to failure, the semantics of this `exhale` never *evaluates* the division, because failure is already reached in the first conjunct. Moreover, for most Viper users, reporting that there is insufficient permission is more intuitive, because Viper users think of the `exhale` as traversing the assertion from left to right, which matches the semantics formalised in this chapter.

The following example illustrates the second downside (unsoundness):

```
inhale acc(x.f)
exhale acc(x.f) && (perm(x.f) == none ==> 0/0 == 0/0)
```

Suppose this statement is executed in a state without any permissions. Then, the `exhale` fails in our formalised semantics, because `perm(x.f)` evaluates to zero (since the permission to `x.f` has been removed by the first conjunct) and thus the right-hand side of the implication is evaluated, which fails due to the division by 0. However, the prior encoding generated a Boogie program that is correct, which means the encoding is unsound. The reason is that the prior encoding checked well-definedness of all expressions *without* taking the removal of permissions into account. That means, in the prior well-definedness check encoding, `perm(x.f)` evaluates to 1 and thus since the left-hand side evaluates to false, the right-hand side is never evaluated. Thus, the generated Boogie program is correct. In summary, the prior encoding did not correctly reflect that even for well-definedness checks, the meaning of a permission introspection instance evaluated during an `exhale` depends on the permissions removed up to that point.

To fix both of these issues, we changed the implementation of `exhale` to include the well-definedness checks on-the-fly [124]. Moreover, we added a flag to the Scala method implementing this functionality to allow switching off the well-definedness checks entirely (*e.g.* for the optimised method call encoding).

[124]: Parthasarathy (2023), *Include definedness checks during exhale (Pull Request 457)*

For the prior implementation of `inhale` such issues did not arise. However, the implementation had separate Scala methods for encoding the `inhale` with well-definedness checks and without well-definedness checks. We merged these implementations and also added a flag [127], which significantly reduced duplication and made the code more understandable. Although this could have been done before, the simplification became clear due to insights regarding interactions with well-definedness checks, as uncovered by our work.

[127]: Parthasarathy (2022), *Move well-definedness checks during inhale into InhaleModule and use new terminology (Pull Request 407)*

3.8.2. Simple and Effective Encoding Improvements

Well-definedness check ordering

Prior to our work, the implementation checked the well-definedness of certain expressions in an unintuitive order. For instance, for a field access

$e.f$, the implementation first emitted Boogie code to check whether there is permission to $e.f$ followed by Boogie code to check whether e is well-defined. This led to unintuitive error messages. For instance, for the expression `x.f.f` in a state without any permissions, the verifier reported that there was insufficient permission to `x.f.f`. This error message is unintuitive because the location `x.f.f` has no meaning, since `x.f` itself is ill-defined. A more intuitive error message is to report that there is insufficient permission to `x.f`. We fixed this by changing the order of well-definedness checks and did so also in other similar cases [128, 129]. This adjustment also simplifies validation, because the semantics formalised in this chapter first checks well-definedness e before checking whether there is sufficient permission to $e.f$.

[128]: Parthasarathy (2023), *Refactor well-definedness of field acesses and permission division (Pull Request 451)*
[129]: Parthasarathy (2022), *Well definedness checking order fixes (Pull Request 429)*

As part of this fix, we also simplified the code. Not every subexpression of the form $e.f$ in the AST needs to be checked to be well-defined. For instance, in **acc**$(e.f, p)$, the field access $e.f$ need not be checked to be well-defined (e and p need to be checked to be well-defined). The prior implementation distinguished these cases using global state that was hard to understand. In our fix, we eliminated this global state and instead used code that is easier to follow. This elimination of global state also made it easier to first check well-definedness of e before checking whether there is sufficient permission to $e.f$. While this simplification and the changing of the well-definedness check order could in principle have been made along, our investigation into the implementation uncovered the corresponding issues.

An extra optimisation

As we discussed in Subsection 3.3.3 on page 127, the Viper-to-Boogie implementation encodes the **`exhale`** of a method call precondition and the **`inhale`** of a method call postcondition in an optimised fashion: well-definedness checks for expressions are omitted, which is justified by the specification well-formedness checks emitted by the implementation for each method. We noticed that the optimisation could go even further: for accessibility predicates, the nonnegativity check on the corresponding permission amount can be omitted as well, because well-formedness of the specification guarantees that these never fail. We have not yet implemented this optimisation, but our certification approach could be used to ensure that this optimisation is indeed sound.

3.8.3. Self-Framing Predicates

We will now discuss the soundness bug that we found outside of our formalised subset. To present this bug, we first need to present some background on *predicate definitions* in Viper, which are inspired by *abstract predicates* originally introduced for separation logic [130]. Predicate definitions in Viper (which are not part of our formalised subset) are top-level definitions, where the body of each such definition is a Viper assertion. Predicates represent permissions abstractly via a predicate instance. A common use case of predicates is to represent permissions to a statically unbounded number of heap locations via recursive predicate definitions (*e.g.* permissions to all the heap locations of a linked list).

[130]: Parkinson et al. (2005), *Separation logic and abstraction*

To keep things simple, we will consider only non-recursive predicate definitions. An example for such a non-recursive predicate definition is:

```
predicate P(x: Ref) {
    acc(x.f) && x.f > 0
}
```

With this definition, one can use the predicate instance `acc(P(x))` as a Viper assertion (*e.g.* as part of a Viper method specification or an `inhale` or `exhale`). `acc(P(x))` essentially *guards* the predicate body assertion. More generally, Viper treats predicates *isorecursively* [107]. That is, Viper treats predicate instances as *opaque* resources in the Viper state; the permissions and heap information specified in the corresponding predicate bodies are not *directly* accessible. For instance, reading or writing to `x.f` would fail even if `acc(P(x))` were held in the state with the predicate definition given above. However, Viper has a primitive statement that makes the assertion guarded by the predicate instance directly accessible: Viper's `unfold` statement exchanges a predicate instance for its body. Moreover, Viper has a primitive statement that guards the predicate body within a predicate instance: Viper's `fold` statement exchanges the predicate body for its corresponding predicate instance. An `unfold` roughly corresponds to exhaling the predicate instance and then inhaling the predicate body, and a `fold` roughly corresponds to exhaling the predicate body and then inhaling the predicate instance. An isorecursive predicate approach with explicit `unfold` and `fold` statements is standard for automated separation logic verifiers such as Viper in order to avoid heuristics for how to inspect predicate definitions.

[107]: Summers et al. (2013), *A Formal Semantics for Isorecursive and Equirecursive State Abstractions*

Viper also supports *fractional amounts* of predicate resources [123]. For instance, `acc(P(x),1/2)` guards half of the permissions specified by `P(x)`, which means that `acc(P(x),1/2)` guards the assertion `acc(x.f ,1/2) && x.f > 0` with the definition given above.

[123]: Dardinier et al. (2022), *Fractional resources in unbounded separation logic*

Let us now focus on the key requirement that Viper imposes on predicate definitions, which forms the source of the soundness bug. Viper requires that predicate bodies specified in predicate definitions are *self-framing*.[36] Intuitively, this means that the assertion itself contains all the permissions needed for the heap locations its validity depends on. The intuition for this requirement is that Viper needs the following property: if a predicate instance is held in a state, then changes to heap locations *not* guarded by the predicate instance should *not* affect the validity of the instance's guarded predicate body. Otherwise, held predicate instance could be invalidated by field assignments, since it would be possible to obtain full permission to a heap location whose evaluation affects the validity of the corresponding predicate body. (As we will see below, Viper's current self-framing requirement is not strong enough to guarantee this property for all predicate instances, which is the reason for the soundness bug.) In our formal semantics, one can express this requirement formally by saying that inhaling the assertion cannot fail starting from any state.[37]

36: The notion of self-framing assertions is common for implicit dynamic frames [33, 101].

37: This is how we, for instance, defined the well-formedness of a method specification in Definition 3.3.1 on page 129.

The problem that we discovered is the following. In general, if a specified predicate body A is self-framing, then this does *not* imply that each assertion representing some *fractional* amount of A is self-framing, too. Thus, the body guarded by a fractional predicate instance may not be self-framing. As a result, the needed property does not always hold for

```
field f: Int
field g: Int

predicate R(x: Ref, y: Ref) {
    acc(x.f) && acc(y.f) && (x == y ==> y.g > 0)
}

method main(a: Ref)
    requires acc(a.g) && acc(a.f)
    ensures  false
{
    a.g := 1
    fold acc(R(a,a),1/2)
    a.g := -1
    unfold acc(R(a,a),1/2)
}
```

Figure 3.30: A Viper program showing why the requirement that predicate bodies must be self-framing is not strong enough and leads to unsoundness.

fractional predicate instances, which leads to the soundness bug. The following predicate definition provides an example:

```
predicate R(x: Ref, y: Ref) {
    acc(x.f) && acc(y.f) && (x == y ==> y.g > 0)
}
```

This predicate body is self-framing and thus accepted by Viper. The reason the body is self-framing even though the right-hand side of the implication contains a heap location to which no permission is specified is the following. Inhaling the first two conjuncts must lead to a normal or magic outcome, and never to failure, since the receivers and permission amounts are always trivially well-defined (and the permission amounts are never negative). If a normal outcome is reached after inhaling the first two conjuncts, then `x` and `y` must be different, otherwise a magic outcome would be reached since the permission mask would be inconsistent. As a result, the left-hand side of the implication evaluates to false, and thus the right-hand side of the implication is never evaluated, which makes the body self-framing. However, the following assertion, which is guarded by the predicate instance `acc(R(x,y),1/2)`, is *not* self-framing:

```
acc(x.f,1/2) && acc(y.f,1/2) && (x == y ==> y.g > 0)
```

Inhaling this assertion in a state with no permissions and where `x` equals `y` leads to failure. In this case, a normal outcome is reached after inhaling the first two conjuncts, and the right-hand side of the implication is evaluated, which leads to failure since there is no permission to `y.g`.

The question that remains is whether this problem leads to an unsoundness. That is, is there a program successfully verified by the Viper verifier, which should not be successfully verified? The answer is yes, and one such program is shown in Figure 3.30. The Viper verifier verifies the program successfully, but it should not. First, let us discuss why the program should not be verified successfully and then let us discuss why the verifier successfully verifies the program. The program should not be verified, because it has a single method `main` with a satisfiable precondition and an unsatisfiable postcondition (*i.e.* **false**). Moreover, `main` only performs (1) heap assignments, which always result in normal outcomes, and (2) a **fold** statement, which just shields previously held permissions behind a predicate instance and an **unfold** statement, which essentially just reverses the previous **fold** statement. So, neither of these two points should lead to magic outcome, and thus there should be a failing method execution.

Why does the Viper verifier successfully verify the program? The **fold** operation succeeds, because there is sufficient permission to evaluate `a.g` in the predicate body and `a.g` is greater than 0 due to the previous assignment. So, here a fractional predicate is successfully folded, whose (fractional) body is not self-framing. The **unfold** then leads to a magic outcome, because from the corresponding **inhale** of the body we learn that `a.g` is larger than 0, but because of the previous assignment, we also know that `a.g` must be -1. As a result, the Viper verifier successfully verifies the program.

The core reason why (fractional) predicate instances held in a Viper state must guard self-framing bodies is because Viper frames all these predicate instances around operations performed on the state (that is, the predicate instances are left unchanged). Self-framedness guarantees that such operations cannot change information held within predicate instances. In the shown program, because the folded predicate instance does not guard a self-framing body, the second assignment is able to change information held within the folded predicate instance. A natural solution would be for Viper to require that *every fraction* of specified predicate bodies in predicate definitions is self-framing (instead of just the fraction 1). However, this requirement may be too strong. For instance, this requirement is not necessary if the fractions used for predicate instances in a concrete Viper program are always greater or equal to 1. In particular, this requirement rules out certain predicates that are potentially useful with the current Viper language. As a result, this requirement would disallow these predicates even if they were used only with fractions that are greater or equal to 1. For example, if a predicate body uses *Viper function applications* whose preconditions require specific amounts of permission, then fractional amounts of the body will not be self-framing. There has been a longstanding discussion (prior to our work) on eliminating function preconditions with specific permission amounts for other reasons. The soundness bug discussed here provides another strong motivation for doing so. In summary, a solution to the problem will require answering broader design questions on predicates (and potentially other features such as functions).

How we discovered the unsoundness

The reason we discovered the previously unknown unsoundness with predicates is the following. We were looking into how the Viper-to-Boogie implementation treats predicate because one of our goals was to make sure that our certification approach could be extended to other Viper features, and predicates are a core feature outside of our subset. The Viper-to-Boogie implementation uses an optimised translation for **inhale** and **exhale** operations of the predicate body as part of the translation for **fold** and **unfold** operations.

This optimisation omits well-definedness checks for expressions in the corresponding assertions, which is analogous to the optimised method call translation. We initially thought that this optimisation for `fold` and `unfold` was justified by the fact that the implementation emitted well-formedness checks for the predicate body, which would be analogous to the justification for the optimised method call translation. However, while trying to prove this justification, we noticed that the well-formedness check was not strong enough to justify the optimisation for fractional predicates. Ultimately, we noticed that there is even an unsoundness if the Viper-to-Boogie implementation does not use an optimised encoding for `fold` and `unfold`, as our example demonstrates (*i.e.* even without the optimisation, the generated Boogie program for Figure 3.30 is correct).

3.9. Future Work

In this section, we discuss some avenues for future work.

3.9.1. Extend Supported Viper Subset

Viper supports commonly used features that are not in our supported subset. The most important features not included in our subset are (1) loops, (2) quantifiers, (3) (labelled) old expressions, which evaluate expressions in a previous Viper state, (4) more-complex resource assertions, which include predicates, magic wands, and iterated separating conjunctions, (5) heap-dependent functions, and (6) domains, which are used to axiomatise constructs in Viper. Extending our Viper semantics with these features and then applying (and if necessary extending) our methodology in order to support these features is one possible future direction.

Describing the semantics of loops and supporting them in the generation of certificates is straightforward: their semantics can be desugared via their invariant, in a pattern similar to method calls that we already support. Extending support to Viper expressions with universal and existential quantification should be straightforward, too. The semantics of quantification can be defined similarly to the semantics of quantification defined for Boogie in Chapter 2; the corresponding generation of certificates should be straightforward. For (labelled) old expressions, preliminary work taken by a student project supervised by us suggests that an extension should be fairly straightforward using the existing methodology. Defining the semantics of the more-complex resource assertions is more involved. Once the semantics is defined, we are confident that the general methodology developed in this thesis for generating certificates will be applicable for these resource assertions. One challenge for these assertions is to formalise a semantics that accurately captures the intended behaviour. While the semantics of predicates, magic wands, and iterated separating conjunctions is well-understood in traditional separation logic settings, formalising their semantics in combination with all of Viper's features still has open challenges. We will discuss one challenge for predicates in more detail below. For heap-dependent functions in an implicit dynamic frames setting, Summers and Drossopoulou [107]

[107]: Summers et al. (2013), *A Formal Semantics for Isorecursive and Equirecursive State Abstractions*

provide a semantics that could be extended to the Viper setting. Our initial exploration into functions suggests that it may be useful to use a generalised form of forward simulations; the main ideas of our methodology should still apply, see Subsection 3.9.2 for a discussion on working with other simulations. Domains without any type parameters should not pose any significant challenges, while formalising the semantics for domains with type parameters may be more challenging.

Finally, we also do not formalise support for unstructured control flow (Viper supports gotos). The semantics of unstructured control flow in Viper has never been formalised. Doing so is not straightforward, because if there are loops, then a Viper semantics needs to take invariants into account to describe the semantics of loops accurately. Identifying loops in unstructured programs is nontrivial (as we saw for Boogie's cycle elimination transformation in Chapter 2). It might make most sense if the semantics gets the loop information as an input (*e.g.* indicating where to loops are) to simplify the semantics. Finally, one would have to tweak parts of our methodology, which currently expects structured Viper programs.

Let us now take a closer look at one of the challenges that arises when formalising Viper predicates (see Subsection 3.8.3 on page 191 for an introduction to predicates). In order to model Viper predicates accurately, a Viper semantics must treat predicates isorecursively (*i.e.* a predicate instance is differentiated from its body), which reflects how the Viper verifier treats predicates and as a result is fundamental for giving an accurate meaning to certain Viper features such as permission introspection. For instance, `perm(x.f)` should *not* take permissions into account that are guarded within a held predicate instance, and `perm(P(x))` should evaluate to the total permission amount held for predicate instance `P(x)` (which is an amount that is independent from the directly held permissions to heap locations). As a result, a Viper state's permission mask must track the directly accessible permissions to heap locations separately from the permission amount held for each predicate instance.

A question that arises with this change to the Viper state is whether and how to extend the consistency of Viper states. Recall that the semantics formalised in this chapter ensures that states remain consistent. In particular, an `inhale` only results in a normal outcome if the resulting state is consistent; if the `inhale` does not fail and the resulting state would be inconsistent, then `inhale` goes to magic. It turns out that to justify the existing Viper-to-Boogie translation for predicates, extending the consistency of Viper states is one possible option (as we will argue below). In our current semantics without predicates, a Viper state is consistent if there is at most one permission to each heap location. An extension that could justify the existing Viper-to-Boogie translation is that the same is true *after* completely unfolding the body of each held predicate instance (recursively until there are no predicate instances left). For example, consider the following predicate definition:

```
predicate P(x: Ref) {
    acc(x.f, 1/2)
}
```

Suppose a Viper state contains both a predicate instance `P(x)` *and* full permission to `x.f`. Then, the state would *not* be consistent (according

```
field f: Int

predicate P(x: Ref) {
    acc(x.f, 1/2)
}

function getVal(x: Ref) : Int
    requires P(x)
{
    unfolding P(x) in x.f
}

method m(x: Ref)
    requires P(x) && acc(x.f) && x.f == 0
    ensures  false
{
    // assert getVal(x) == 0
    x.f := x.f + 1
    // assert getVal(x) == 1
}
```

Figure 3.31: Example showing why we consider an extended notion of consistency and why this extended notion justifies the existing Viper-to-Boogie translation. The Boogie program, which is generated by the existing Viper-to-Boogie translation for this Viper program, is correct. For the verifier to report success in practice, the `assert` statements must be uncommented to trigger quantifiers in the Boogie encoding.

to the extended notion), since after completely unfolding `P(x)`, there is more than one permission to `x.f`. That means, after inhaling `acc(P(x)) && acc(x.f)` using the definition of `P(x)` above, the magic outcome would be reached with this extended consistency notion.

Summers and Drossopoulou [107] have defined such an extended notion of consistency in order to formally show the relation of an isorecursive predicate semantics with an *equirecursive* predicate semantics. In an equirecursive semantics, a predicate instance and its body are not differentiated. Instead, predicates are directly interpreted via their least fixed points. There are multiple challenges that are not addressed by Summers and Drossopoulou [107], which would need to be solved to extend our work for Viper using such an extended notion of consistency. First, the extended notion of consistency would be fundamentally part of the (isorecursive) Viper semantics, instead of just being used for proving a formal relation with an equirecursive semantics. Second, since Viper has *iterated separating conjunctions* potentially ranging over an infinite domain, a Viper state may hold infinitely many predicate instances. As a result, formally defining the extended notion of consistency must essentially compute *infinite sums*. Third, since Viper supports existential quantification of permission amounts via *wildcards*, there is not a uniquely determined permission amount for each heap location even after completely unfolding each predicate instance.

[107]: Summers et al. (2013), *A Formal Semantics for Isorecursive and Equirecursive State Abstractions*

Let us now discuss why it even makes sense to consider an extended notion of consistency as part of the Viper semantics in order to justify the existing Viper-to-Boogie translation (instead of using the original consistency definition). This is not obvious, because the existing Viper-to-Boogie translation does not inspect the bodies of predicates explicitly, except as part of **`fold`** and **`unfold`** statements.

The Viper program shown in Figure 3.31 justifies an extended notion of consistency. This example contains the same predicate definition `P` that we used before and additionally contains a *heap-dependent function* `getVal`, which returns the value of `x.f`.[38] `getVal` specifies the predicate instance `P(x)` in its precondition in order to ensure that the evaluation of `x.f` is well-defined. The **`unfolding`** expression instructs the verifier to use the body of `P(x)` to justify the well-definedness of `x.f`.

38: The body of heap-dependent functions are always just expressions and thus they have no side effects.

The Boogie program, which is generated by the existing Viper-to-Boogie translation for this Viper program, is correct (we will discuss why in the next paragraph). That is, to justify the Viper-to-Boogie translation, the Viper semantics must ensure that this Viper program, which has **`false`** as a method postcondition, is correct. Using the extended notion of consistency, the Viper program is indeed correct, because the state reached after inhaling the precondition of `m` is always inconsistent, and thus every execution goes to magic. The reason is that the permission amount to `x.f` contained in the body of `P(x)` and the precondition's second conjunct together exceed 1. This behaviour is completely natural for an equirecursive semantics, which does not distinguish between a predicate instance and its body (in an equirecursive semantics, the assertion **`acc`**`(x .f) && P(x)` cannot be satisfied). However, for an isorecursive semantics such as the one we are considering for Viper, this behaviour is not obvious. Next, we will make clear why we are considering this behaviour in the first place: because the generated Boogie program is correct.

Finally, let us discuss why the Boogie program generated by the existing Viper-to-Boogie translation for this Viper program is correct. The reason is that the encoding used by the translation for this program essentially makes explicit that the predicate instance `P(x)` before and after the field assignment must be the same one. Moreover, the encoding reflects that `getVal` depends only on those heap locations to which `P(x)` specifies permissions. Thus, the program encodes that the value returned by `getVal` must be the same before and after the heap assignment, because the predicate instance before and after the field assignment is the same one. As a result, there is a contradiction, because the value returned by `getVal` (*i.e.* `x.f`) is different before and after the field assignment, and thus the postcondition **`false`** holds in the encoding.[39]

39: In practice, the Boogie verifier will not be able to derive the contradiction, because the necessary quantifiers in the Boogie axioms are not triggered. If one uncomments the **`assert`** statements shown in Figure 3.31 (which both succeed in the Viper semantics), then the Boogie verifier will derive the contradiction for the resulting Boogie program. Note that an **`assert`** statement that succeeds does not change the semantics of a Viper program.

At a high level, the Boogie encoding frames Viper predicate instances around statements, and this fact can be explicitly observed in the encoding of Viper functions, without requiring any **`unfold`** or **`fold`** statements in the corresponding Viper program. The extended consistency notion justifies this by ensuring that if the heap is updated, then it is guaranteed that no predicate instance is affected (if additionally the predicate bodies guarded by predicate instances are self-framing, see Subsection 3.8.3 on page 191).

Once the semantics of predicates is defined, one can apply our methodology to extend the generation of certificates to support predicates. One important aspect for certification is to include the extended state consistency into the state relations tracked by simulation proofs in the certificate. To achieve this, one will have to prove that the Viper semantics (extended to predicates) preserves the extended state consistency. Moreover, one will have to formally show that the extended state consistency can indeed be used to justify the existing Viper-to-Boogie translation.

3.9.2. Extension to Other Simulations

Our methodology in this chapter uses forward simulations as the underlying simulation. An advantage of forward simulations is that they are simple to reason about. However, forward simulations cannot be used to justify every possible translation. In some cases, different simulations are required. One direction for future work is to apply the main concepts from our methodology to other kinds of simulations. The main concepts from our methodology are (1) splitting a simulation syntactically and semantically into smaller simulations, (2) handling diverse translations by sufficiently parameterising simulations, and (3) propagating properties implied by non-local checks in the state relation in systematic ways. None of these concepts are specifically tied to forward simulations. We chose forward simulations for our use case, because it is a simple kind of simulation that is sufficient to reason about a variety of front-end translations. In particular, we demonstrated the applicability of forward simulations on the existing Viper-to-Boogie implementation for our subset. Moreover, as part of an internship project supervised by us, Bonneau [63] showed that the existing Dafny-to-Boogie translation could also be reasoned about using forward simulations.

[63]: Bonneau (2021), *A formal foundation for the Dafny verifier*

In some cases, when forward simulations are not expressive enough for translations, one can split the reasoning into two parts, one of which can be handled by forward simulations, and the other must be handled with a different simulation. One example in our work is the encoding of scoped Viper variables into Boogie. The existing Viper-to-Boogie translation just declares a unique local Boogie variable (declared at the start of the Boogie method) for each scoped Viper variable. The encoding of the scoped variable itself does not result in any Boogie command in the procedure body. This encoding cannot be justified using only a forward simulation, because in the Viper semantics the value for the scoped variable is chosen nondeterministically at the point when the corresponding declaration is executed within the Viper method body, while in the Boogie procedure the value for the corresponding local variable l is chosen *at the beginning* of the procedure. For every Viper execution, a forward simulation tracks a *single* value for l in the Boogie state until the Viper scoped declaration is executed. Since it is impossible to choose the correct value up front that is going to be chosen by the scoped declaration, the forward simulation cannot justify the translation. As discussed in Section 3.6, our workaround is to introduce a **havoc** statement for l in the Boogie procedure at the point when the scoped Viper variable is introduced, which allows choosing the value for l at the same point as in Viper, and thus is justifiable using a forward simulation. One would need to separately show that the correctness of the original Boogie encoding implies the correctness of the Boogie encoding with the **havoc** statements using a different kind of simulation. Forward simulations are also not expressive enough to handle Boogie's assignment elimination presented in Chapter 2 for a similar reason. In the validation of assignment elimination, we handled the entire transformation using a different kind of simulation (tracking a *set* of Boogie states instead of a single Boogie state for each Viper state). This different simulation can be seen as a generalised forward simulation.

3.9.3. Front-End Translations that Encode Program Logics

Some front-end translations encode program logics into the IVL. For instance, various front-end translations into Viper encode a flavour of concurrent separation logic into Viper (see Section 3.7 for examples). In such translations, a parallel composition of two statements in the source program is typically encoded into three IVL procedures: one procedure per parallel branch, each of which is verified using a separate specification provided by the user, and one for the enclosing code that contains the parallel branch, which composes the two specifications to encode the behaviour of the parallel composition overall. This translation encodes concurrent separation logic's rule for parallel compositions.

In work not presented in this dissertation, we show that a natural way of justifying such translations is to use an *axiomatic semantics* of the input language (*i.e.* the front-end program logic) and an *axiomatic semantics* of the IVL [45]. In this setting, soundness is shown by proving that a valid derivation for the IVL program in its axiomatic semantics implies a valid derivation for the input program in its axiomatic semantics. In contrast, the work in this chapter works directly at the level of an operational semantics for both the input language and the IVL, which is natural for front-end translations that encode an operational semantics into the IVL instead of encoding a program logic (such as the Viper-to-Boogie translation). It depends on the translation whether working with an axiomatic semantics or an operational semantics is more convenient.

[45]: Dardinier et al. (2025), *Formal Foundations for Translational Separation Logic Verifiers*

In our work on soundness proofs that use an axiomatic semantics for the input language and IVL [45], we consider once-and-for-all proofs (instead of per-run validation) and we do not consider actual implementations used in practice. One direction for future work would be to develop a per-run validation approach for automatically generating certificates for front-end translations implemented in practice that encode program logics. To do so, one could port ideas developed in this chapter to the setting where soundness proofs use an axiomatic semantics for the input language and the IVL.

3.9.4. Leveraging Syntactic Checks on the Boogie Code

As we discussed in Subsection 3.5.2 on page 157, we use the auxiliary variable map in our state relation instantiation to prove that certain Boogie variables are not modified during the simulation of some Viper effect. For example, the simulation of **`remcheck`** A is justified via the judgement $\mathsf{rcSim}_{\Gamma_b}(R, R', A, \gamma, \gamma')$. To prove that Boogie executions justifying this judgement do not modify certain Boogie variables, we reflect these variables explicitly in the auxiliary variable maps of R and R'. It would be more convenient if we could instead just syntactically check that the Boogie code simulating the Viper effect does not modify the variables in question. This would simplify working with simulation rules such as the method call rule (see Figure 3.21 on page 151), where we must show that certain variables are not modified during the simulation of the corresponding **exhale** and **inhale** operations. However, such a syntactic check is not straightforward here, because one would need to overapproximate the set of commands executed by Boogie executions starting from program point γ and ending in γ'. One direction for future work would be to adjust the approach to make such syntactic checks straightforward with the goal of simplifying certificates.

One way to potentially achieve this would be to connect the Viper AST to an intermediate Boogie AST representation, which has the same structure as the Viper AST. In a separate step, one would connect the intermediate Boogie AST representation with the actual Boogie AST used by the Boogie verifier.[40] For instance, a Viper sequential composition $s_1; s_2$ would be encoded in the intermediate AST by a Boogie sequential composition $t_1; t_2$ where t_i encodes s_i.[41] This way one could more easily identify, which Boogie statement captures which Viper effect (*e.g.* t_i captures the execution of s_i), and could thus use syntactic checks.

40: The Viper-to-Boogie implementation internally uses such an intermediate Boogie AST representation that has the same structure as the Viper AST representation.

41: Such a Boogie sequential composition does not exist in the Boogie AST representation that we target.

Advantage of current approach

A potential downside of using such an intermediate Boogie AST representation is that one may lose generality. In particular, our current approach, which directly targets the Boogie AST as represented by the Boogie verifier, is essentially agnostic towards what representation is used for the Boogie program. One could, for instance, replace the Boogie AST semantics by the Boogie control-flow graph semantics defined in Chapter 2 in our simulation relation, and everything would essentially still work the same way. For instance, in an internship project supervised by us, Bonneau [63] showed that one can use our current approach to directly connect a Dafny program represented as an AST with Boogie's control-flow graph representation. This is likely harder to achieve using an approach that relies on the structural similarity of the source and target representations.

3.9.5. End-to-End Certificates

Given a Viper program P_v, our certificate-producing Viper-to-Boogie implementation generates a certificate connecting the Viper program P_v with the corresponding Boogie program P_b. In order to apply our certificate-producing Boogie verifier discussed in Chapter 2 to P_b in order to produce a certificate connecting P_b with the corresponding verification condition, there are still some extensions required. Once these extensions have been implemented, one could connect the two certificates to obtain an end-to-end certificate that connects P_v directly with Boogie's verification condition. The required extensions are the following. First, certificate-producing support needs to be added for the features discussed in Subsection 3.3.5 on page 131. Second, the certificate optimisations and the incompletenesses discussed in Subsection 2.12.1 on page 91 in Chapter 2, which are relevant for Viper-generated Boogie programs, must be implemented. Finally, the dead variable elimination of the Boogie verifier needs to also be certified in case the Viper-generated Boogie program leads to Boogie variables that are eliminated.

This concludes this chapter on the formal validation of the existing Viper-to-Boogie implementation. Our work makes the existing implementation certificate-producing, thus significantly increasing the implementation's trustworthiness. As part of doing this work, we were able to positively impact the Viper ecosystem by fixing bugs and developing a deeper understanding of the Viper language.

Conclusion 4.

We have presented formal translation validation approaches for translational program verifiers. We have applied these approaches to the existing Boogie and Viper verifier implementations. This application, enabled via an instrumentation of the existing implementations, led to the automatic production of a formal certificate on every verifier run such that Isabelle can check these certificates automatically. As a result, our work demonstrates that it is feasible to provide strong formal guarantees for verification results of existing and practical translational program verifiers written in mainstream programming languages.

Our work has had impact beyond demonstrating the feasibility of formal guarantees for existing and practical implementations. For instance, we have discovered soundness bugs and made general improvements as a result of our work in Viper's ecosystem (Section 3.8 discusses some of these). Moreover, exploratory work on formally reasoning about the Dafny-to-Boogie translation via our Boogie semantics [63] discovered that if polymorphic maps were extensional, then the Dafny-to-Boogie translation would be unsound [131].[1] At the time, Boogie did not emit extensionality axioms for polymorphic maps, and thus there was no soundness issue in practice. However, such a change could easily be integrated into Boogie. Many of these discoveries are hard to identify without deeply thinking about the implemented translations and the formal semantics of the languages involved, which is something we were required to do for the work presented by this dissertation.

[63]: Bonneau (2021), *A formal foundation for the Dafny verifier*

[131]: Parthasarathy (2022), *Soundness of Dafny relies on Boogie maps not being extensional (Issue 2463)*

1: The Dafny GitHub issue was posted by the author of this dissertation, but Benjamin Bonneau found the potential unsoundness.

More broadly, our work has enabled answering or identifying questions. For instance, when Boogie developers were working on extending their monomorphisation approach to polymorphic maps and type quantification in 2022 (which was eventually merged [99]), we were able to support them by formally explaining what a feasible formal semantics for these features is, using our work as evidence. For type quantification, our work provides a formal semantics and demonstrates that Boogie's generated verification condition respects this semantics. Our work does not provide a formal semantics for polymorphic maps in general, but our work shows how to formally capture instances of polymorphic maps used in practice (*e.g.* to represent heaps), which helped gain a deeper understanding for their intended meaning. Moreover, our goal of putting Viper on a more formal footing has helped with supporting design decisions and has helped identify questions that need to be answered (Subsection 3.8.3 on page 191 discusses one example). The impact on Viper via formal foundations has been a collaborative effort, which has been achieved through different projects. This dissertation contributed to this impact significantly. Moreover, the work led by Thibault Dardinier, which is not presented in this dissertation but to which the author of this dissertation also contributed, also had a significant impact. The combined insights gained from these collaborations were important to gain a clearer understanding of large parts of the Viper ecosystem.

[99]: Qadeer (2022), *Monomorphization of polymorphic maps and binders*

As future work, there are many directions one could take. We have outlined some concrete future directions in Section 2.12 on page 91 and Section 3.9 on page 195. We end this dissertation by discussing a broader future direction. In particular, the automatic certificate production infrastructures, which we have added to existing implementations, are research prototypes. We have maintained these research prototypes as part of forks of the main codebase. There are still questions to be answered, in order to turn such research prototypes into mature infrastructures that are part of the main codebase, and which are actively maintained by verifier developers. Next, we discuss such questions as part of three dimensions: (1) the scalability to large input programs, (2) the formal semantics of the languages involved in translations, and (3) the maintainability of the certificate production infrastructure.

Scalability

One question is to what extent our high-level approach scales to large programs. We have demonstrated our approach on small to medium-sized programs, which is a substantial improvement over not having any formal guarantees at all. However, to increase the likelihood that such an approach is accepted by verifier developers, one must be able to handle most programs in reasonable time, which includes large programs. That means, Isabelle needs to be able to check the generated certificates in reasonable time, which requires certificate optimisations. We have discussed promising optimisations in Subsection 2.12.1 on page 91 (for instance, we discuss an optimisation for a quadratic overhead for Boogie variables, which leads to slow certificate checking times for Boogie programs with very many variables). More work needs to be done and optimisations must be implemented to demonstrate that our approach scales to all programs that show up in practice.

Formal semantics

A requirement of our approach is that one must have a formal semantics for the languages involved, otherwise it is not possible to provide formal guarantees. However, for some languages, the semantics of certain features is essentially an open research problem. In our case, we considered languages (Boogie and Viper) that have substantially fewer features than mainstream programming languages, and even for those, formalising the semantics is challenging for certain features of the language. Also, we considered only subsets of these languages, which capture substantial parts of the languages, but are subsets nonetheless. As a result, two natural questions arise. First, how does the approach scale to larger and more complex language subsets? To deal with this question, we have taken care to design a modular approach via our systematic decomposition into smaller problems. Our decomposition allows one to deal with different aspects of the language independently. Nevertheless, one must still demonstrate that our approach indeed scales to larger and more complex language subsets. Second, if one only has a formal semantics for a subset of a language, how can one substantiate the benefit of the work? For the second question, we have evidence of the positive impact the work can have on a language and verifier ecosystem despite supporting only a subset of the language (some examples were discussed in Section 3.8).

Maintainability

In order for the certificate production to be part of the main code base, one must answer the question of how to make the production of certificates maintainable over time. Certificate production requires the instrumentation of the implementation to provide hints, and requires that the proof automation works as expected. For instance, if a large part of the translation changes substantially, then there is potentially nontrivial work involved in adjusting both the instrumentation and the proof automation. Our approach tries to minimise this work as much as possible. For instance, using our approach, if only one feature is affected then in many cases one need not adjust the certification of other features. However, there are changes that could globally affect all features. It would be beneficial if the certificate production were set up such that even for such substantial changes it is clear how the different components need to be adjusted at a high level. It might also be in general useful to provide debugging tools for the case when a certificate is not successfully checked by Isabelle. For new verifier features or substantial changes to the translation, one may want to introduce a temporary time frame where certificates are not supported for particular features or where a certificate with additional assumptions is produced. This would allow for verifier development to move faster, while allowing the certificate production code to remain part of the main code base.

While these questions still need to be answered, this dissertation demonstrates a significant improvement in terms of formally establishing the soundness of translational program verifier *implementations used in practice*. We hope that in the future the research community will further expand on this line of work such that it becomes the norm that translational program verifier implementations used in practice enjoy formal soundness guarantees.

Appendix A.

A.1. Detailed Results of the Evaluation in Chapter 3

The detailed results of the evaluation presented in Section 3.6 on page 177 are shown separately for each of the files in Table 1.1 (Gobra), Table 1.2 (MPP), Table 1.3 (VerCors), and Table 1.4 (Viper).

Table 1.1: Detailed results of our evaluation for the files from the test suite of Gobra.

File	**Methods** no.	**Viper** Total [LoC]	**Boogie** Total [LoC]	**Isabelle** Total [LoC]	**Proof Check** Total [s]
concurrency	2	24	164	1153	25.3
defer-simple-01	6	142	639	3344	49.6
defer-simple-02	9	211	853	4717	60.6
perm-fail1	15	165	661	6392	66.5
perm-simple1	9	131	622	4221	50.7
fail1	3	44	283	1574	31.6
fail3	2	19	116	1044	23.6
simple1	2	30	237	1210	28.4
simple2	1	10	90	672	21.4
simple3	1	17	186	801	24.6
global-const-8	6	49	206	2510	33.8
pointer-identity	1	30	158	731	23.0
pointer-identity	1	30	158	731	23.1
000008	1	10	85	672	21.4
000009	1	16	98	679	21.3
000039	3	49	178	1410	26.8
000155	2	39	152	1075	24.3

Table 1.2: Detailed results of our evaluation for the files from the test suite of MPP.

File	**Methods** no.	**Viper** Total [LoC]	**Boogie** Total [LoC]	**Isabelle** Total [LoC]	**Proof Check** Total [s]
banerjee	8	414	2014	9545	242.4
darvas	2	91	582	2800	38.4
kusters	3	112	583	3146	46.2

Table 1.3: Detailed results of our evaluation for the files from the test suite of VerCors.

File	**Methods** no.	**Viper** Total [LoC]	**Boogie** Total [LoC]	**Isabelle** Total [LoC]	**Proof Check** Total [s]
BasicAssert-e1	6	41	197	2589	35.0
BasicAssert	6	41	193	2589	35.0
DafnyIncr	8	60	265	3419	41.6
DafnyIncrE1	8	57	220	3340	40.2
permissions	5	39	208	2270	33.1
inv-test-fail1	5	90	510	2589	55.5
inv-test-fail2	5	92	514	2596	56.5
inv-test	5	90	510	2589	55.1
SwapIntegerFail	8	79	429	3645	49.8
SwapIntegerPass	8	81	469	3688	53.0
SwapLong	6	57	277	2725	36.7
SwapLongTwice	8	81	469	3688	52.1
SwapLongWrong	8	79	429	3645	48.9
frame-error-1	5	35	173	2191	32.8
refute3	6	49	246	2662	34.5
refute4	6	54	258	2676	35.9
refute5	6	50	253	2662	35.9
demo1	7	60	347	3185	44.6

Table 1.4: Detailed results of our evaluation for the files from the test suite of Viper.

File	**Methods** no.	**Viper** Total [LoC]	**Boogie** Total [LoC]	**Isabelle** Total [LoC]	**Proof Check** Total [s]
0004	1	6	100	729	21.7
0004-CPG1	1	6	95	704	21.6
0005	1	4	78	665	21.1
0008	2	12	241	1396	26.8
0011	5	63	902	3284	55.7
0015	1	6	92	709	21.4
0052	1	7	100	719	21.5
0063	6	34	180	2595	35.2
0072	1	8	112	770	22.4
0073	1	10	132	737	22.2
0088-1	1	9	115	751	21.9
0094	1	6	91	679	21.2
0152	2	14	139	1137	24.5
0157	8	47	354	3508	45.1
0159	2	13	120	1083	23.8
0170	1	8	84	665	21.1
0177-1	1	10	102	665	21.4
0222	2	13	118	1054	23.9
0227	1	5	85	683	21.4
0324	1	7	104	704	21.2
0345	3	21	165	1463	24.4
0384	1	11	127	709	22.0
assert	1	7	92	693	21.5
negative-amounts	3	21	155	1517	27.4
old	6	38	318	2805	37.9
swap	2	16	177	1239	25.6
test	1	6	81	663	20.9
testHistoryProcesses	13	205	1711	7035	126.3
testHistoryProcessesPVL	13	204	1711	7035	116.3
testHistoryProcessesPVL-CPG1	4	56	490	2304	46.1
testHistoryThreadsProcessesPVL	4	56	490	2304	45.7
test-example1	4	57	374	2152	37.0
test-example3	5	74	430	2634	39.3
test-example4	5	71	451	2645	42.7

Bibliography

Here are the references in citation order.

[1] K. Rustan M. Leino. 'This is Boogie 2'. Available from http://research.microsoft.com/en-us/um/people/leino/papers/krml178.pdf. 2008 (cited on pages 2, 3, 9, 24, 91, 97).

[2] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 'Creusot: A Foundry for the Deductive Verification of Rust Programs'. In: *International Conference on Formal Engineering Methods (ICFEM)*. Ed. by Adrián Riesco and Min Zhang. Vol. 13478. 2022, pp. 90–105. DOI: 10.1007/978-3-031-17244-1_6 (cited on pages 2, 3, 97, 98).

[3] K. Rustan M. Leino. 'Dafny: An Automatic Program Verifier for Functional Correctness'. In: *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. Ed. by Edmund M. Clarke and Andrei Voronkov. 2010. DOI: 10.1007/978-3-642-17511-4_20 (cited on pages 2, 3, 14, 97, 177).

[4] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 'Frama-C: A software analysis perspective'. In: *Formal Aspects of Computing* 27.3 (2015), pp. 573–609. DOI: 10.1007/s00165-014-0326-7 (cited on pages 2, 4, 97).

[5] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João Carlos Pereira, and Peter Müller. 'Gobra: Modular Specification and Verification of Go Programs'. In: *Computer Aided Verification (CAV)*. Ed. by Alexandra Silva and K. Rustan M. Leino. 2021. DOI: 10.1007/978-3-030-81685-8_17 (cited on pages 2, 4, 99, 178).

[6] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. 'The VerCors Tool Set: Verification of Parallel and Concurrent Software'. In: *Integrated Formal Methods (IFM)*. Ed. by Nadia Polikarpova and Steve Schneider. 2017. DOI: 10.1007/978-3-319-66845-1_7 (cited on pages 2, 4, 99, 178).

[7] P. Müller, M. Schwerhoff, and A. J. Summers. 'Viper: A Verification Infrastructure for Permission-Based Reasoning'. In: *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. 2016. DOI: 10.1007/978-3-662-49122-5_2 (cited on pages 2, 4, 9, 14, 97).

[8] Sascha Böhme and Tjark Weber. 'Fast LCF-Style Proof Reconstruction for Z3'. In: *Interactive Theorem Proving (ITP)*. Ed. by Matt Kaufmann and Lawrence C. Paulson. 2010. DOI: 10.1007/978-3-642-14052-5_14 (cited on pages 2, 4, 13, 89, 97).

[9] Burak Ekici, Alain Mebsout, Cesare Tinelli, Chantal Keller, Guy Katz, Andrew Reynolds, and Clark W. Barrett. 'SMTCoq: A Plug-In for Integrating SMT Solvers into Coq'. In: *Computer Aided Verification (CAV)*. Ed. by Rupak Majumdar and Viktor Kuncak. 2017. DOI: 10.1007/978-3-319-63390-9_7 (cited on pages 2, 4, 13, 89, 97).

[10] Mathias Fleury and Hans-Jörg Schurr. 'Reconstructing veriT Proofs in Isabelle/HOL'. In: *Workshop on Proof eXchange for Theorem Proving (PxTP)*. Ed. by Giselle Reis and Haniel Barbosa. 2019. DOI: 10.4204/EPTCS.301.6 (cited on pages 2, 4, 13, 89, 97).

[11] The Coq Development Team. *The Coq Reference Manual – Release 8.19.0*. https://coq.inria.fr/doc/V8.19.0/refman. 2024 (cited on page 2).

[12] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*. Vol. 2283. Lecture Notes in Computer Science. Springer, 2002 (cited on pages 2, 13, 97).

[13] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 'The Lean Theorem Prover (System Description)'. In: *Conference on Automated Deduction (CADE)*. Ed. by Amy P. Felty and Aart Middeldorp. Vol. 9195. Lecture Notes in Computer Science. Springer, 2015, pp. 378–388. DOI: 10.1007/978-3-319-21401-6_26 (cited on page 2).

[14] Chris Lattner and Vikram S. Adve. 'LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation'. In: *Code Generation and Optimization (CGO)*. IEEE Computer Society, 2004, pp. 75–88. DOI: 10.1109/CGO.2004.1281665 (cited on page 3).

[15] Akash Lal and Shaz Qadeer. 'Powering the static driver verifier using corral'. In: *Foundations of Software Engineering (FSE)*. Ed. by Shing-Chi Cheung, Alessandro Orso, and Margaret-Anne D. Storey. ACM, 2014, pp. 202–212. DOI: 10.1145/2635868.2635894 (cited on pages 3, 97).

[16] Montgomery Carter, Shaobo He, Jonathan Whitaker, Zvonimir Rakamaric, and Michael Emmi. 'SMACK software verification toolchain'. In: *International Conference on Software Engineering (ICSE)*. Ed. by Laura K. Dillon, Willem Visser, and Laurie A. Williams. ACM, 2016, pp. 589–592. DOI: 10.1145/2889160.2889163 (cited on pages 3, 14, 97).

[17] J.-C. Filliâtre and A. Paskevich. 'Why3 — Where Programs Meet Provers'. In: *European Symposium on Programming (ESOP)*. Ed. by Matthias Felleisen and Philippa Gardner. 2013. DOI: 10.1007/978-3-642-37036-6_8 (cited on pages 4, 89, 97).

[18] Marco Eilers and Peter Müller. 'Nagini: A Static Verifier for Python'. In: *Computer Aided Verification (CAV)*. Ed. by Hana Chockler and Georg Weissenbacher. 2018. DOI: 10.1007/978-3-319-96145-3_33 (cited on pages 4, 99).

[19] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 'Leveraging Rust Types for Modular Specification and Verification'. In: vol. 3. OOPSLA. Association for Computing Machinery, 2019. DOI: 10.1145/3360573 (cited on pages 4, 97, 99).

[20] John C. Reynolds. 'Separation logic: A logic for shared mutable data structures'. In: *Logic in Computer Science (LICS)*. 2002, pp. 55–74. DOI: 10.1109/lics.2002.1029817 (cited on pages 4, 99).

[21] Cormac Flanagan and James B. Saxe. 'Avoiding exponential explosion: generating compact verification conditions'. In: *Principles of Programming Languages (POPL)*. Ed. by Chris Hankin and Dave Schmidt. 2001. DOI: 10.1145/360204.360220 (cited on pages 4, 50).

[22] K. Rustan M. Leino. 'Efficient weakest preconditions'. In: *Information Processing Letters* 93.6 (2005), pp. 281–288. DOI: https://doi.org/10.1016/j.ipl.2004.10.015 (cited on pages 4, 50, 90).

[23] Michael Barnett and K. Rustan M. Leino. 'Weakest-precondition of unstructured programs'. In: *Workshop on Program Analysis For Software Tools and Engineering (PASTE)*. Ed. by Michael D. Ernst and Thomas P. Jensen. 2005. DOI: 10.1145/1108792.1108813 (cited on pages 4, 13, 23, 50, 90).

[24] Jean Fortin. 'BSP-Why, a tool for deductive verification of BSP programs: machine-checked semantics and application to distributed state-space algorithms'. PhD thesis. University of Paris-Est, France, 2013 (cited on page 5).

[25] Paolo Herms. 'Certification of a Tool Chain for Deductive Program Verification'. PhD thesis. University of Paris-Sud, Orsay, France, 2013 (cited on pages 5, 182).

[26] Muhammad Taimoor Khan. 'Formal Specification and Verification of Computer Algebra Software'. PhD thesis. Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, 2014 (cited on page 5).

[27] Hermann Lehner and Peter Müller. 'Formal Translation of Bytecode into BoogiePL'. In: *Electronic Notes in Theoretical Computer Science* 190.1 (2007). Workshop on Bytecode Semantics, Verification, Analysis and Transformation (Bytecode 2007), pp. 35–50. DOI: https://doi.org/10.1016/j.entcs.2007.02.059 (cited on pages 5, 182).

[28] Frédéric Vogels, Bart Jacobs, and Frank Piessens. 'A Machine Checked Soundness Proof for an Intermediate Verification Language'. In: *Theory and Practice of Computer Science, Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM)*. Ed. by Mogens Nielsen, Antonín Kucera, Peter Bro Miltersen, Catuscia Palamidessi, Petr Tuma, and Frank D. Valencia. Vol. 5404. Lecture Notes in Computer Science. Springer, 2009, pp. 570–581. DOI: 10.1007/978-3-540-95891-8_51 (cited on pages 5, 182).

[29] Michael Backes, Cătălin HriŢcu, and Thorsten Tarrach. 'Automatically Verifying Typing Constraints for a Data Processing Language'. In: *Certified Programs and Proofs (CPP)*. Ed. by Jean-Pierre Jouannaud and Zhong Shao. 2011. DOI: https://doi.org/10.1007/978-3-642-25379-9_22 (cited on pages 5, 182).

[30] Cyrill Martin Gössi. 'A Formal Semantics for Viper'. Master's Thesis. ETH Zurich, 2016 (cited on pages 5, 182, 187).

[31] Frédéric Vogels, Bart Jacobs, and Frank Piessens. 'A machine-checked soundness proof for an efficient verification condition generator'. In: *SAC*. 2010 (cited on pages 5, 90, 91).

[32] Peter V. Homeier and David F. Martin. 'A Mechanically Verified Verification Condition Generator'. In: *The Computer Journal* 38.2 (1995), pp. 131–141 (cited on pages 5, 90).

[33] Jan Smans, Bart Jacobs, and Frank Piessens. 'Implicit Dynamic Frames'. In: *Transactions on Programming Languages and Systems (TOPLAS)* 34.1 (May 2012). DOI: 10.1145/2160910.2160911 (cited on pages 5, 98, 103, 129, 188, 192).

[34] Lionel Blatter, Nikolai Kosmatov, Virgile Prevosto, and Pascale Le Gall. 'Certified Verification of Relational Properties'. In: *Integrated Formal Methods (IFM)*. Ed. by Maurice H. ter Beek and Rosemary Monahan. Vol. 13274. Lecture Notes in Computer Science. Springer, 2022, pp. 86–105. DOI: 10.1007/978-3-031-07727-2_6 (cited on pages 5, 90).

[35] Quentin Garchery. 'A Framework for Proof-carrying Logical Transformations'. In: *Workshop on Proof eXchange for Theorem Proving (PxTP)*. Ed. by Chantal Keller and Mathias Fleury. 2021. DOI: 10.4204/EPTCS.336.2 (cited on pages 5, 89, 97, 185).

[36] Joshua M. Cohen and Philip Johnson-Freyd. 'A Formalization of Core Why3 in Coq'. In: *Proc. ACM Program. Lang.* 8.POPL (2024). DOI: 10.1145/3632902 (cited on pages 5, 90, 185).

[37] Jean-Baptiste Tristan and Xavier Leroy. 'Formal verification of translation validators: a case study on instruction scheduling optimizations'. In: *Principles of Programming Languages (POPL)*. Ed. by George C. Necula and Philip Wadler. 2008. DOI: 10.1145/1328438.1328444 (cited on pages 6, 14, 184).

[38] Jean-Baptiste Tristan and Xavier Leroy. 'Verified validation of lazy code motion'. In: *Programming Language Design and Implementation (PLDI)*. Ed. by Michael Hind and Amer Diwan. 2009. DOI: 10.1145/1542476.1542512 (cited on pages 6, 184).

[39] Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby Murray, Gabriele Keller, and Gerwin Klein. 'A Framework for the Automatic Formal Verification of Refinement from Cogent to C'. In: *ITP*. 2016 (cited on pages 6, 91).

[40] Jeehoon Kang, Yoonseung Kim, Youngju Song, Juneyoung Lee, Sanghoon Park, Mark Dongyeon Shin, Yonghyun Kim, Sungkeun Cho, Joonwon Choi, Chung-Kil Hur, and Kwangkeun Yi. 'Crellvm: verified credible compilation for LLVM'. In: *Programming Language Design and Implementation (PLDI)*. Ed. by Jeffrey S. Foster and Dan Grossman. ACM, 2018, pp. 631–645. DOI: 10.1145/3192366.3192377 (cited on pages 6, 184).

[41] Léo Gourdin, Benjamin Bonneau, Sylvain Boulmé, David Monniaux, and Alexandre Bérard. 'Formally Verifying Optimizations with Block Simulations'. In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (2023). DOI: 10.1145/3622799 (cited on pages 6, 184).

[42] Andreas Lööw, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Caroline Cronjäger, Petar Maksimovic, and Philippa Gardner. 'Compositional Symbolic Execution for Correctness and Incorrectness Reasoning (Extended Version)'. In: *CoRR* abs/2407.10838 (2024). DOI: 10.48550/arXiv.2407.10838 (cited on page 6).

[43] Frédéric Vogels, Bart Jacobs, and Frank Piessens. 'Featherweight VeriFast'. In: *Log. Methods Comput. Sci.* 11.3 (2015). DOI: 10.2168/LMCS-11(3:19)2015 (cited on page 6).

[44] Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. 'Sound Gradual Verification with Symbolic Execution'. In: *Proc. ACM Program. Lang.* 8.POPL (2024), pp. 2547–2576. DOI: 10.1145/3632927 (cited on pages 6, 188).

[45] Thibault Dardinier, Michael Sammler, Gaurav Parthasarathy, Alexander J. Summers, and Peter Müller. 'Formal Foundations for Translational Separation Logic Verifiers'. In: *Proc. ACM Program. Lang.* 9.POPL (2025). DOI: 10.1145/3704856 (cited on pages 6, 12, 123, 182, 185, 186, 188, 200).

[46] Stefan Wils and Bart Jacobs. 'Certifying C program correctness with respect to CH2O with VeriFast'. In: *CoRR* abs/2308.15567 (2023). DOI: 10.48550/ARXIV.2308.15567 (cited on pages 6, 90, 185).

[47] Norman D. Megill and David A. Wheeler. *Metamath: A Computer Language for Mathematical Proofs*. 2019. URL: http://us.metamath.org/downloads/metamath.pdf (cited on page 6).

[48] Zhengyao Lin, Xiaohong Chen, Minh-Thai Trinh, John Wang, and Grigore Rosu. 'Generating Proof Certificates for a Language-Agnostic Deductive Program Verifier'. In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (2023), pp. 56–84. DOI: 10.1145/3586029 (cited on pages 6, 90, 185).

[49] Steven Keuchel, Sander Huyghebaert, Georgy Lukyanov, and Dominique Devriese. 'Verified symbolic execution with Kripke specification monads (and no meta-programming)'. In: *Proc. ACM Program. Lang.* 6.ICFP (2022), pp. 194–224. DOI: 10.1145/3547628 (cited on page 7).

[50] Andrew W. Appel. 'VeriSmall: Verified Smallfoot Shape Analysis'. In: *Certified Programs and Proofs (CPP)*. Ed. by Jean-Pierre Jouannaud and Zhong Shao. Vol. 7086. Lecture Notes in Computer Science. Springer, 2011, pp. 231–246. DOI: 10.1007/978-3-642-25379-9_18 (cited on page 7).

[51] Adam Chlipala. 'Mostly-automated verification of low-level programs in computational separation logic'. In: *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. Ed. by Mary W. Hall and David A. Padua. ACM, 2011, pp. 234–245. DOI: 10.1145/1993498.1993526 (cited on page 7).

[52] Ike Mulder, Robbert Krebbers, and Herman Geuvers. 'Diaframe: automated verification of fine-grained concurrent programs in Iris'. In: *PLDI '22: 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. Ed. by Ranjit Jhala and Isil Dillig. ACM, 2022, pp. 809–824. DOI: 10.1145/3519939.3523432 (cited on page 7).

[53] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. 'RefinedC: automating the foundational verification of C code with refined ownership types'. In: *Programming Language Design and Implementation (PLDI)*. Ed. by Stephen N. Freund and Eran Yahav. ACM, 2021, pp. 158–174. DOI: 10.1145/3453483.3454036 (cited on page 7).

[54] Lennard Gäher, Michael Sammler, Ralf Jung, Robbert Krebbers, and Derek Dreyer. 'RefinedRust: A Type System for High-Assurance Verification of Rust Programs'. In: *Proc. ACM Program. Lang.* 8.PLDI (2024), pp. 1115–1139. DOI: 10.1145/3656422 (cited on page 7).

[55] Qinxiang Cao, Lennart Beringer, Samuel Gruetter, Josiah Dodds, and Andrew W. Appel. 'VST-Floyd: A Separation Logic Tool to Verify Correctness of C Programs'. In: *J. Autom. Reason.* 61.1-4 (2018), pp. 367–422. DOI: 10.1007/S10817-018-9457-5 (cited on page 7).

[56] Pierre-Yves Strub, Nikhil Swamy, Cédric Fournet, and Juan Chen. 'Self-certification: bootstrapping certified typecheckers in F* with Coq'. In: *Principles of Programming Languages (POPL)*. Ed. by John Field and Michael Hicks. ACM, 2012. DOI: 10.1145/2103656.2103723 (cited on pages 7, 89).

[57] Aymeric Fromherz, Aseem Rastogi, Nikhil Swamy, Sydney Gibson, Guido Martínez, Denis Merigoux, and Tahina Ramananandro. 'Steel: proof-oriented programming in a dependently typed concurrent separation logic'. In: *Proc. ACM Program. Lang.* 5.ICFP (2021), pp. 1–30. DOI: 10.1145/3473590 (cited on page 7).

[58] Nikhil Swamy, Aseem Rastogi, Aymeric Fromherz, Denis Merigoux, Danel Ahman, and Guido Martínez. 'SteelCore: an extensible concurrent separation logic for effectful dependently typed programs'. In: *Proc. ACM Program. Lang.* 4.ICFP (2020), 121:1–121:30. DOI: 10.1145/3409003 (cited on page 7).

[59] Gaurav Parthasarathy, Peter Müller, and Alexander J. Summers. 'Formally Validating a Practical Verification Condition Generator'. In: *Computer Aided Verification (CAV)*. Ed. by Alexandra Silva and K. Rustan M. Leino. Vol. 12760. LNCS. 2021, pp. 704–727. DOI: 10.1007/978-3-030-81688-9_33 (cited on pages 11, 87).

[60] Gaurav Parthasarathy, Thibault Dardinier, Benjamin Bonneau, Peter Müller, and Alexander J. Summers. 'Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language'. In: *Proc. ACM Program. Lang.* 8.PLDI (2024). DOI: 10.1145/3656438 (cited on pages 11, 102, 117).

[61] Aleksandar Hubanov. 'Formally Validating the AST-to-CFG Phase of the Boogie Program Verifier'. Bachelor's Thesis. ETH Zurich, 2022 (cited on pages 11, 84).

[62] Lukas Himmelreich. 'Formally Validating the CFG Optimization Phase of the Boogie Program Verifier'. Bachelor's Thesis. ETH Zurich, 2023 (cited on pages 11, 74).

[63] Benjamin Bonneau. *A formal foundation for the Dafny verifier*. Internship Report. 2021 (cited on pages 11, 199, 201, 203).

[64] Thibault Dardinier, Gaurav Parthasarathy, and Peter Müller. 'Verification-Preserving Inlining in Automatic Separation Logic Verifiers'. In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (2023), pp. 789–818. DOI: 10.1145/3586054 (cited on pages 12, 188).

[65] Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J. Summers. 'Sound Automation of Magic Wands'. In: *Computer Aided Verification (CAV)*. Ed. by Sharon Shoham and Yakir Vizel. Vol. 13372. Lecture Notes in Computer Science. Springer, 2022, pp. 130–151. DOI: 10.1007/978-3-031-13188-2_7 (cited on pages 12, 188).

[66] Ralf Jung, Rodolphe Lepigre, Gaurav Parthasarathy, Marianna Rapoport, Amin Timany, Derek Dreyer, and Bart Jacobs. 'The future is ours: prophecy variables in separation logic'. In: *Proc. ACM Program. Lang.* 4.POPL (2020), 45:1–45:32. DOI: 10.1145/3371113 (cited on page 12).

[67] K. Rustan M. Leino and Philipp Rümmer. 'A Polymorphic Intermediate Verification Language: Design and Logical Encoding'. In: *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Ed. by Javier Esparza and Rupak Majumdar. 2010. DOI: 10.1007/978-3-642-12002-2_26 (cited on pages 14, 19, 24, 35, 36, 62, 63, 101, 133).

[68] Xavier Leroy. 'Formal certification of a compiler back-end or: programming a compiler with a proof assistant'. In: *POPL*. 2006 (cited on pages 14, 91).

[69] Gilles Barthe, Delphine Demange, and David Pichardie. 'Formal Verification of an SSA-Based Middle-End for CompCert'. In: *TOPLAS* 36.1 (2014) (cited on pages 14, 91).

[70] Akash Lal, Shaz Qadeer, and Shuvendu K. Lahiri. 'A Solver for Reachability Modulo Theories'. In: *Computer Aided Verification (CAV)*. Ed. by P. Madhusudan and Sanjit A. Seshia. 2012. DOI: 10.1007/978-3-642-31424-7_32 (cited on page 14).

[71] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 'SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs'. In: *Computer Aided Verification (CAV)*. Ed. by P. Madhusudan and Sanjit A. Seshia. Vol. 7358. Lecture Notes in Computer Science. Springer, 2012, pp. 712–717. DOI: 10.1007/978-3-642-31424-7_54 (cited on pages 14, 97).

[72] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, Michał Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. 'VCC: A Practical System for Verifying Concurrent C'. In: *Theorem Proving in Higher Order Logics (TPHOLs)*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. 2009. DOI: 10.1007/978-3-642-03359-9_2 (cited on page 14).

[73] Gaurav Parthasarathy. *Boogie Semantics and Certificate Metatheory Formalisation*. Accessed December 9, 2024. 2024. URL: https://github.com/viperproject/foundational-boogie/tree/90411340ac568c7870e85dd9ec627b84f01e79a3 (cited on pages 15, 24).

[74] Nancy A. Lynch and Frits W. Vaandrager. 'Forward and Backward Simulations: I. Untimed Systems'. In: *Inf. Comput.* 121.2 (1995), pp. 214–233. DOI: 10.1006/inco.1995.1134 (cited on pages 21, 99, 138).

[75] Boogie Developers. *Boogie implementation*. Accessed May 13, 2024. URL: https://github.com/boogie-org/boogie (cited on page 24).

[76] Matthew S. Hecht and Jeffrey D. Ullman. 'Flow Graph Reducibility'. In: *SIAM J. Comput.* 1.2 (1972), pp. 188–202 (cited on page 37).

[77] Matthew S. Hecht and Jeffrey D. Ullman. 'Characterizations of Reducible Flow Graphs'. In: *J. ACM* 21.3 (1974), pp. 367–375. DOI: 10.1145/321832.321835 (cited on page 37).

[78] K. Rustan M. Leino, Todd D. Millstein, and James B. Saxe. 'Generating error traces from verification-condition counterexamples'. In: *Science of Computer Programming* 55.1-3 (2005), pp. 209–226 (cited on page 63).

[79] Aaron Tomb. *Add missing antecedents to function axioms (Pull Request 749)*. Accessed April 9, 2024. June 2023. URL: https://github.com/boogie-org/boogie/pull/749 (cited on page 65).

[80] Andrew W. Appel and Sandrine Blazy. 'Separation Logic for Small-Step cminor'. In: *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007, Kaiserslautern, Germany, September 10-13, 2007, Proceedings*. Ed. by Klaus Schneider and Jens Brandt. Vol. 4732. Lecture Notes in Computer Science. Springer, 2007, pp. 5–21. DOI: 10.1007/978-3-540-74591-4_3 (cited on page 81).

[81] Xavier Leroy. 'A Formally Verified Compiler Back-end'. In: *J. Autom. Reason.* 43.4 (2009), pp. 363–446. DOI: 10.1007/S10817-009-9155-4 (cited on page 81).

[82] Vladimir Klebanov, Peter Müller, Natarajan Shankar, Gary T. Leavens, Valentin Wüstholz, Eyad Alkassar, Rob Arthan, Derek Bronish, Rod Chapman, Ernie Cohen, Mark Hillebrand, Bart Jacobs, K. Rustan M. Leino, Rosemary Monahan, Frank Piessens, Nadia Polikarpova, Tom Ridge, Jan Smans, Stephan Tobies, Thomas Tuerk, Mattias Ulbrich, and Benjamin Weiß. 'The 1st Verified Software Competition: Experience Report'. In: *FM*. 2011 (cited on page 86).

[83] YuTing Chen and Carlo A. Furia. 'Triggerless Happy – Intermediate Verification with a First-Order Prover'. In: *iFM*. 2017 (cited on page 86).

[84] Boogie Developers. *Boogie Verifier Test Suite Used For Evaluation*. https://github.com/boogie-org/boogie/tree/b4be7f72e3c74cfa9257f385e2c59613b8ced898/Test (cited on page 85).

[85] Jacques-Henri Jourdan, François Pottier, and Xavier Leroy. 'Validating LR(1) Parsers'. In: *European Symposium on Programming (ESOP)*. Ed. by Helmut Seidl. 2012. DOI: 10.1007/978-3-642-28869-2_20 (cited on page 88).

[86] Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, and Thomas Sewell. 'Candle: A Verified Implementation of HOL Light'. In: *Interactive Theorem Proving (ITP)*. Ed. by June Andronick and Leonardo de Moura. 2022 (cited on page 88).

[87] Alejandro Aguirre. *Towards a Provably Correct Encoding from F* to SMT*. Tech. rep. INRIA, 2016 (cited on page 89).

[88] Clark Barrett, Leonardo de Moura, and Pascal Fontaine. 'Proofs in Satisfiability Modulo Theories'. In: *All about Proofs, Proofs for All*. Vol. 55. Mathematical Logic and Foundations. College Publications, 2015, pp. 23–44 (cited on page 90).

[89] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. 'VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java'. In: *NASA Formal Methods (NFM)*. 2011, pp. 41–55. DOI: 10.1007/978-3-642-20398-5_4 (cited on pages 90, 188).

[90] Paolo Herms, Claude Marché, and Benjamin Monate. 'A Certified Multi-prover Verification Condition Generator'. In: *VSTTE*. 2012 (cited on page 90).

[91] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. 'A formally-verified C static analyzer'. In: *POPL*. 2015 (cited on page 90).

[92] Ryan Doenges, Tobias Kappé, John Sarracino, Nate Foster, and Greg Morrisett. 'Leapfrog: certified equivalence for protocol parsers'. In: *Programming Language Design and Implementation (PLDI)*. Ed. by Ranjit Jhala and Isil Dillig. ACM, 2022, pp. 950–965. DOI: 10.1145/3519939.3523715 (cited on page 90).

[93] Eyad Alkassar, Sascha Böhme, Kurt Mehlhorn, and Christine Rizkallah. 'A Framework for the Verification of Certifying Computations'. In: *JAR* 52.3 (2014), pp. 241–273 (cited on page 90).

[94] Jean-Baptiste Tristan and Xavier Leroy. 'Formal Verification of Translation Validators: A Case Study on Instruction Scheduling Optimizations'. In: *POPL*. 2008 (cited on page 91).

[95] Jean-Baptiste Tristan and Xavier Leroy. ‘A simple, verified validator for software pipelining’. In: *POPL*. 2010 (cited on page 91).

[96] Sandrine Blazy, Delphine Demange, and David Pichardie. ‘Validating Dominator Trees for a Fast, Verified Dominance Test’. In: *ITP*. 2015 (cited on page 91).

[97] Radu Grigore. ‘The Design and Algorithms of a Verification Condition Generator’. PhD thesis. University College Dublin, 2012 (cited on page 91).

[98] seL4 Developers. *Efficient lookup table creation in Isabelle*. Accessed May 11, 2024. URL: https://github.com/seL4/l4v/blob/eb3db4bf34b3e7584093f6e8e95503659a12351d/tools/c-parser/StaticFun.thy (cited on pages 93, 180).

[99] Shaz Qadeer. *Monomorphization of polymorphic maps and binders*. Accessed March 19, 2024. Dec. 2022. URL: https://github.com/boogie-org/boogie/pull/669 (cited on pages 94, 95, 136, 188, 203).

[100] Ioannis T. Kassios. ‘Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions’. In: *Formal Methods (FM)*. Ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. 2006. DOI: 10.1007/11813040_19 (cited on page 98).

[101] Matthew J. Parkinson and Alexander J. Summers. ‘The Relationship Between Separation Logic and Implicit Dynamic Frames’. In: *Logical Methods in Computer Science* 8.3:01 (2012), pp. 1–54. DOI: 10.2168/LMCS-8(3:1)2012 (cited on pages 98, 103, 129, 192).

[102] Viper Developers. *Viper-to-Boogie implementation (https://github.com/viperproject/carbon)*. Accessed April 4, 2024. 2024. URL: https://github.com/viperproject/carbon (cited on page 99).

[103] Gaurav Parthasarathy. *Viper Semantics and Certificate Metatheory Formalisation*. Accessed December 9, 2024. 2024. URL: https://github.com/viperproject/viper-roots/tree/845f8eed90c6dd51fad779 (cited on pages 102, 109).

[104] Gaurav Parthasarathy, Thibault Dardinier, Benjamin Bonneau, Peter Müller, and Alexander J. Summers. *Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language – Artifact*. 2024. DOI: 10.5281/zenodo.10802176 (cited on page 102).

[105] John Boyland. ‘Checking Interference with Fractional Permissions’. In: *Static Analysis (SAS)*. Ed. by Radhia Cousot. 2003, pp. 55–72. DOI: 10.1007/3-540-44898-5_4 (cited on page 103).

[106] David Detlefs, Greg Nelson, and James B. Saxe. ‘Simplify: a theorem prover for program checking’. In: *J. ACM* 52.3 (2005), pp. 365–473. DOI: 10.1145/1066100.1066102 (cited on page 134).

[107] Alexander J. Summers and Sophia Drossopoulou. ‘A Formal Semantics for Isorecursive and Equirecursive State Abstractions’. In: *European Conference on Object-Oriented Programming (ECOOP)*. Ed. by Giuseppe Castagna. Vol. 7920. Lecture Notes in Computer Science. Springer, 2013, pp. 129–153. DOI: 10.1007/978-3-642-39038-8_6 (cited on pages 170, 192, 195, 197).

[108] Marco Eilers, Peter Müller, and Samuel Hitz. ‘Modular Product Programs’. In: *European Symposium on Programming (ESOP)*. Ed. by Amal Ahmed. 2018. DOI: 10.1007/978-3-319-89884-1_18 (cited on page 178).

[109] K. Rustan M. Leino and Peter Müller. ‘A Basis for Verifying Multi-threaded Programs’. In: *European Symposium on Programming (ESOP)*. Ed. by Giuseppe Castagna. Vol. 5502. Lecture Notes in Computer Science. Springer, 2009, pp. 378–393. DOI: 10.1007/978-3-642-00590-9_27 (cited on pages 182, 187).

[110] K. Rustan M. Leino, Peter Müller, and Jan Smans. ‘Verification of Concurrent Programs with Chalice’. In: *Foundations of Security Analysis and Design V, FOSAD 2007/2008/2009 Tutorial Lectures*. Ed. by Alessandro Aldini, Gilles Barthe, and Roberto Gorrieri. Vol. 5705. Lecture Notes in Computer Science. Springer, 2009, pp. 195–222. DOI: 10.1007/978-3-642-03829-7_7 (cited on pages 182, 187).

[111] Alexander J. Summers and Peter Müller. ‘Automating deductive verification for weak-memory programs (extended version)’. In: *International Journal on Software Tools for Technology Transfer (STTT)* 22.6 (2020), pp. 709–728. DOI: 10.1007/S10009-020-00559-Y (cited on page 182).

[112] Felix A. Wolf, Malte Schwerhoff, and Peter Müller. ‘Concise outlines for a complex logic: a proof outline checker for TaDA’. In: *Formal Methods in System Design* 61.1 (2022), pp. 110–136. DOI: 10.1007/S10703-023-00427-W (cited on page 182).

[113] Claude Marché and Yannick Moy. *The Jessie plugin for Deductive Verification in Frama-C*. 2018. URL: http://krakatoa.lri.fr/jessie.pdf (cited on page 182).

[114] Christine Rizkallah, Japheth Lim, Yutaka Nagashima, Thomas Sewell, Zilin Chen, Liam O'Connor, Toby C. Murray, Gabriele Keller, and Gerwin Klein. 'A Framework for the Automatic Formal Verification of Refinement from Cogent to C.' In: *Interactive Theorem Proving (ITP)*. Ed. by Jasmin Christian Blanchette and Stephan Merz. 2016. DOI: 10.1007/978-3-319-43144-4_20 (cited on page 184).

[115] Simon Winwood, Gerwin Klein, Thomas Sewell, June Andronick, David A. Cock, and Michael Norrish. 'Mind the Gap'. In: *Theorem Proving in Higher Order Logics (TPHOLS)*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. 2009. DOI: 10.1007/978-3-642-03359-9_34 (cited on page 184).

[116] David A. Cock, Gerwin Klein, and Thomas Sewell. 'Secure Microkernels, State Monads and Scalable Refinement'. In: *Theorem Proving in Higher Order Logics (TPHOLS)*. Ed. by Otmane Ait Mohamed, César A. Muñoz, and Sofiène Tahar. 2008. DOI: 10.1007/978-3-540-71067-7_16 (cited on page 184).

[117] Gerwin Klein, Thomas Sewell, and Simon Winwood. 'Refinement in the Formal Verification of the seL4 Microkernel'. In: *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Ed. by David S. Hardin. Springer, 2010, pp. 323–339. DOI: 10.1007/978-1-4419-1539-9_11 (cited on page 184).

[118] Amir Pnueli, Michael Siegel, and Eli Singerman. 'Translation Validation'. In: *Tools and Algorithms for Construction and Analysis of Systems(TACAS)*. Ed. by Bernhard Steffen. Vol. 1384. Lecture Notes in Computer Science. Springer, 1998, pp. 151–166. DOI: 10.1007/BFB0054170 (cited on page 184).

[119] Martin C. Rinard and Darko Marino. 'Credible Compilation with Pointers'. In: *Workshop on Run-Time Result Verification*. 1999 (cited on page 184).

[120] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 'Translation validation for a verified OS kernel'. In: *Programming Language Design and Implementation (PLDI)*. Ed. by Hans-Juergen Boehm and Cormac Flanagan. ACM, 2013, pp. 471–482. DOI: 10.1145/2491956.2462183 (cited on page 185).

[121] Malte Schwerhoff. 'Advancing Automated, Permission-Based Program Verification Using Symbolic Execution'. PhD thesis. ETH Zurich, Zürich, Switzerland, 2016. DOI: 10.3929/ETHZ-A-010835519 (cited on pages 187, 188).

[122] Jenna DiVincenzo, Ian McCormack, Hemant Gouni, Jacob Gorenburg, Mona Zhang, Conrad Zimmerman, Joshua Sunshine, Éric Tanter, and Jonathan Aldrich. 'Gradual C0: Symbolic Execution for Efficient Gradual Verification'. In: *CoRR* abs/2210.02428 (2022). DOI: 10.48550/ARXIV.2210.02428 (cited on page 188).

[123] Thibault Dardinier, Peter Müller, and Alexander J. Summers. 'Fractional resources in unbounded separation logic'. In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (2022), pp. 1066–1092. DOI: 10.1145/3563326 (cited on pages 188, 192).

[124] Gaurav Parthasarathy. *Include definedness checks during exhale (Pull Request 457)*. Accessed July 18, 2024. Mar. 2023. URL: https://github.com/viperproject/carbon/pull/457 (cited on pages 188, 190).

[125] Gaurav Parthasarathy. *Viper allows predicates whose fractional amount is not self-framing (Issue 809)*. Accessed September 13, 2024. Aug. 2024. URL: https://github.com/viperproject/silver/issues/809 (cited on page 189).

[126] Gaurav Parthasarathy. *The semantics of permission introspection in the body of unfolding expressions is unclear (Issue 682)*. Accessed July 19, 2024. Aug. 2023. URL: https://github.com/viperproject/silver/issues/682 (cited on page 189).

[127] Gaurav Parthasarathy. *Move well-definedness checks during inhale into InhaleModule and use new terminology (Pull Request 407)*. Accessed July 18, 2024. Mar. 2022. URL: https://github.com/viperproject/carbon/pull/407 (cited on page 190).

[128] Gaurav Parthasarathy. *Refactor well-definedness of field acesses and permission division (Pull Request 451)*. Accessed July 18, 2024. Feb. 2023. URL: https://github.com/viperproject/carbon/pull/451 (cited on page 191).

[129] Gaurav Parthasarathy. *Well definedness checking order fixes (Pull Request 429)*. Accessed July 18, 2024. Aug. 2022. URL: https://github.com/viperproject/carbon/pull/429 (cited on page 191).

[130] Matthew J. Parkinson and Gavin M. Bierman. 'Separation logic and abstraction'. In: *Principles of Programming Languages (POPL)*. Ed. by Jens Palsberg and Martín Abadi. ACM, 2005, pp. 247–258. DOI: 10.1145/1040305.1040326 (cited on page 191).

[131] Gaurav Parthasarathy. *Soundness of Dafny relies on Boogie maps not being extensional (Issue 2463)*. Accessed August 15, 2024. July 2022. URL: https://github.com/dafny-lang/dafny/issues/2463 (cited on page 203).