Formal Foundations for Automated Deductive Verifiers

Thibault Dardinier

DISS. ETH NO. 31492

DISS. ETH NO. 31492

FORMAL FOUNDATIONS FOR AUTOMATED DEDUCTIVE VERIFIERS

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES

(Dr. sc. ETH Zurich)

presented by

THIBAULT GABRIEL GILBERT DARDINIER

born on March 26, 1996

accepted on the recommendation of

Prof. Dr. Peter Müller, examiner

Prof. Dr. Gilles Barthe, co-examiner

Prof. Dr. Derek Dreyer, co-examiner

Dr. Nikhil Swamy, co-examiner

2025

Abstract

Automated deductive verifiers are tools that attempt to prove, with mathematical certainty, that all executions of a program satisfy a given specification, using program logics such as Hoare logic or separation logic. Modern verifiers have already had significant impact in industry: the F* verifier has been used at Microsoft to verify code deployed in Microsoft Azure, Firefox, and the Linux kernel; the Gobra verifier (based on Viper) has been used to verify SCION's next-generation router; and the Dafny verifier has been used to verify the core authorization engine of Amazon Web Services, which runs a billion times per second. Despite these achievements, modern verifiers face two key challenges. First, for verification results to be *trustworthy*, verifiers must be sound, *i.e.*, they should only verify programs that actually satisfy their specifications. However, unsoundnesses (cases where a verifier incorrectly verifies an invalid program) are regularly discovered in practice, undermining trust in these tools. Second, automated verifiers are limited in their *expressiveness*. While they can prove properties of individual executions (such as the absence of runtime errors), they fall short when it comes to establishing *hyperproperties*, an important class of functional and security properties that relate multiple executions of a program. This thesis addresses both of these challenges.

To address the trustworthiness challenge, this thesis develops formal foundations for establishing the soundness of automated verifiers based on *separation logic* (SL), a state-of-the-art class of program logics for modular reasoning about sequential and concurrent heap-manipulating programs, and the basis of many modern verifiers. We start by introducing the first formal framework for proving the soundness of SL-based *translational verifiers*, which work by translating the input program and its specification into an intermediate verification language (IVL), subsequently checked by a dedicated verifier for the IVL. Our framework applies to a wide range of translational verifiers, including Gillian (for C, JavaScript, Rust), VeriFast (for C, Java, C++, Rust), and Viper (for C, Java, Rust, Go, Python, and others). Crucially, our framework modularizes the reasoning required for the correctness of the front-end translation from that of the back-end verifier, while supporting diverse verification algorithms and heuristics in the back-end. We demonstrate its practical utility by instantiating it for Viper and connecting it to a front-end translation for concurrent programs. Second, we focus on *fractional predicates*, a generalization of fractional permissions to arbitrary SL predicates, which enable automated verifiers to reason about concurrent reads of shared data structures. We identify a fundamental discrepancy between the theoretical treatment of fractional predicates and their practical implementation in automated verifiers. To resolve this, we present a novel semantics for SL assertions that allows states to temporarily hold more than full permission to a heap location during assertion evaluation. This semantics formally justifies the rules used by existing automated verifiers and provides a foundation for further extensions. Third, we address the automation of the *magic wand* (also called *separating implication*), a key SL connective for reasoning about ownership of partial data structures. Prior to this work, all support for magic wands in automated verifiers was either manual or unsound. We present a novel formal foundation that characterizes the broad design space of sound and automated verification algorithms for magic wands, and use it to implement, in Viper, the first such algorithm.

To address the expressiveness challenge, we introduce Hyper Hoare Logic (HHL), a novel program logic for hyperproperties. HHL generalizes Hoare logic by lifting assertions from predicates over individual states to predicates over sets of states. As a result, HHL can be used to establish a broad range of hyperproperties, encompassing those supported by existing program logics as well as hyperproperties beyond their reach. Despite its expressiveness, we show that HHL admits intuitive and powerful inference rules that capture important reasoning principles, *e.g.*, to compose different types of hyperproperties in the same proof, or to reason about loops where different executions perform different numbers of iterations. We then demonstrate that HHL is amenable to automation by presenting Hypra, a novel automated verifier for hyperproperties based on HHL. Hypra automates HHL by translating an input program and its HHL specification into a Viper program, where one execution of the Viper program simulates a set of executions of the input program. Our evaluation on new and existing benchmarks demonstrates that Hypra can effectively prove a large class of hyperproperties in reasonable time and with minimal annotation overhead.

All formal results in this thesis have been formalized in the interactive proof assistant Isabelle/HOL.

Résumé

Be you, be proud of you, because you can be do what we want to do.

François Hollande

Les *vérificateurs déductifs automatisés* sont des outils dont l'objectif est de construire une preuve mathématique que toutes les exécutions d'un programme donné satisfont une spécification donnée, en s'appuyant sur des logiques de programmes telles que la logique de Hoare ou la logique de séparation. Ces vérificateurs ont déjà eu un impact significatif dans l'industrie : F* a été utilisé par Microsoft pour vérifier du code déployé dans Microsoft Azure, Firefox et le noyau Linux ; Gobra (basé sur Viper) a servi à vérifier le code du routeur de nouvelle génération de SCION ; et Dafny a été utilisé pour vérifier le moteur d'autorisation au cœur d'Amazon Web Services, qui s'exécute un milliard de fois par seconde. Malgré ces succès, ces outils font face à deux défis majeurs. D'abord, pour que les résultats de vérification soient *fiables*, les outils doivent être corrects, c'est-à-dire qu'ils ne doivent valider que des programmes qui satisfont effectivement leurs spécifications. Cependant, des cas d'incorrection (quand un vérificateur valide à tort un programme) sont régulièrement découverts en pratique, ce qui diminue la confiance dans ces outils. Ensuite, les vérificateurs automatisés sont limités dans leur *expressivité*. S'ils peuvent prouver des propriétés d'exécutions individuelles (telles que l'absence d'erreurs à l'exécution), ils s'avèrent insuffisants lorsqu'il s'agit d'établir des *hyperpropriétés*, une classe importante de propriétés fonctionnelles et de sécurité qui relient plusieurs exécutions d'un programme. Cette thèse s'attaque à ces deux défis.

Afin de relever le défi de la fiabilité, cette thèse propose des fondations formelles pour établir la correction de vérificateurs automatisés basés sur la *logique de séparation*. Cette famille de logique de programmes, qui représente l'état de l'art en matière de raisonnement modulaire sur les programmes séquentiels et concurrents, sert de fondement à de nombreux vérificateurs modernes. Nous introduisons d'abord le premier cadre formel pour démontrer la correction des *vérificateurs translationnels* basés sur la logique de séparation, qui fonctionnent en traduisant le programme d'entrée et sa spécification dans un langage intermédiaire de vérification, ensuite vérifié par un vérificateur automatisé dédié. Notre cadre s'applique à un large éventail de vérificateurs translationnels, dont Gillian (pour C, JavaScript, Rust), VeriFast (pour C, Java, C++, Rust) et Viper (pour C, Java, Rust, Go, Python, et d'autres). Fait essentiel, notre cadre modularise le raisonnement requis pour la correction des traductions dans le langage intermédiaire de celui requis pour les vérificateurs dédiés, tout en étant compatible avec une diversité d'algorithmes et d'heuristiques de vérification pour ces derniers. Nous démontrons son utilité pratique en l'instanciant pour Viper et en le reliant à une traduction pour des programmes concurrents en entrée. Deuxièmement, nous nous concentrons sur les *prédicats fractionnaires*, une généralisation des permissions fractionnaires à des prédicats arbitraires de logique de séparation, qui permettent aux vérificateurs automatisés de raisonner sur des lectures concurrentes de structures de données partagées. Nous mettons en évidence un décalage fondamental entre le traitement théorique des prédicats fractionnaires et leur mise en œuvre pratique dans les vérificateurs automatisés. Pour y remédier, nous proposons une nouvelle sémantique des prédicats de logique de séparation qui autorise les états à détenir temporairement plus que la permission pleine sur une cellule du tas pendant l'évaluation des prédicats. Cette sémantique justifie formellement les règles utilisées par les vérificateurs automatisés existants et fournit une base pour de futures extensions. Troisièmement, nous traitons l'automatisation de la *baguette magique* (également appelée *implication séparante*), un connecteur clé de la logique de séparation pour raisonner sur la possession partielle de structures de données Avant cette thèse, tout support de la baguette magique dans les vérificateurs automatisés était soit manuel, soit incorrect. Nous présentons de nouvelles fondations formelles qui caractérisent un large espace de conception d'algorithmes de vérification à la fois corrects et automatisés pour la baguette magique, et nous les utilisons pour développer, pour Viper, le premier algorithme de ce type.

Pour répondre au défi de l'expressivité, nous introduisons *Hyper Hoare Logic* (HHL), une nouvelle logique de

programmes pour les hyperpropriétés. HHL généralise la logique de Hoare en passant de préconditions et postconditions exprimées comme des prédicats sur des états individuels à des prédicats sur des ensembles d'états. En conséquence, HHL supporte un large éventail d'hyperpropriétés, englobant celles prises en charge par les logiques de programmes existantes ainsi que des hyperpropriétés au-delà de leur portée. Malgré son expressivité, nous montrons que HHL admet des règles d'inférence à la fois intuitives et puissantes qui capturent des principes de raisonnement essentiels, par exemple pour composer différents types d'hyperpropriétés au sein d'une même preuve, ou pour raisonner sur des boucles où différentes exécutions effectuent un nombre d'itérations différent. Nous montrons ensuite que HHL se prête à l'automatisation en présentant Hypra, un nouveau vérificateur automatisé pour les hyperpropriétés, basé sur HHL. Hypra automatise HHL en traduisant un programme d'entrée et sa spécification HHL en un programme en Viper, où une exécution de ce dernier simule un ensemble d'exécutions du programme d'entrée. Notre évaluation sur des benchmarks nouveaux et existants montre que Hypra peut prouver efficacement une large classe d'hyperpropriétés en un temps raisonnable, avec un effort d'annotation minimal.

Tous les résultats formels de cette thèse ont été formalisés dans l'assistant de preuve interactif Isabelle/HOL.

Acknowledgements

No sense of urgency.

The Programming Methodology group

Pursuing a PhD over the past five years has been one of the most rewarding experiences of my life, and I am deeply grateful to everyone who has been part of this journey, including many whose names may not appear here.

First and foremost, I want to thank my advisor, Peter Müller. Beyond being an outstanding researcher, you are a genuinely kind person who truly cares about his students. From the very beginning, you believed in me, gave me the freedom I needed to pursue my own ideas (even when you were skeptical), and always found time to help despite leading such a large group. I have always been impressed by how quickly you develop deep intuition about technical problems, and inspired by your long-term research vision. Your constant support, both for my research and my career, has meant a great deal to me, and I know I can always rely on you.

I am grateful to my co-examiners, Gilles Barthe, Derek Dreyer, and Nikhil Swamy, for agreeing to serve on my committee, taking the time to review my thesis, as well as for their insightful questions during my defense. I am additionally grateful to Derek and Nik for their mentorship and support during the faculty job application process.

The most important scientific collaboration of my PhD was with Gaurav Parthasarathy. Gaurav, thank you for being the best MSc thesis supervisor anyone could ask for, the best collaborator I could have hoped for, and for helping me keep my sanity at the start of the PhD. Thank you for the countless discussions we had, whether about early promising ideas, deep technical details, the bangers and the freebies, navigating the PhD, or life in general. I always appreciated the high standard you set for our work, which taught me a lot. I miss our collaboration, but I am excited to see all you will achieve in your professional career.

A highlight of my PhD was being part of the amazing Programming Methodology group. I want to thank all the current members of the PM group, in reverse alphabetical order by last name (following an ancestral Viper tradition), for creating such a stimulating and friendly environment: Thank you, Yushuo, for being the easiest student to supervise; Dionisios, for the amazing padel games; Sandra, for making my life easier in so many ways; João, for all the interesting discussions we had (I especially miss our Seattle summer and the jacuzzi discussions) and for having a *situation*; Hongyi, for losing to me at chess *so many times*; Anqi, for being a perfectly organized student and for the great cakes; Nick, for your deep knowledge of bread and for being the AI token of the group; Jonáš, for making the group a very fun place to be, for organizing so many great events and trips, for being a great VerifyThis teammate, and for introducing me to new music genres (such as acid jazz); Marco, for recognizing my great pronunciation of the letter *h* (I'm still working on the remaining 25 letters. It's true.), for introducing me to Scooter (and hyperproperties as a byproduct), and for being such a fun *pillar* of the group; Lea, for the amazing train ride, for all the *football* games, and for teaching me Waltz; Aurea, for the great chess games, the sushi, and "the Munitch"; and Linard, for the amazing group retreats and for sharing my bed (probably) too many times. I am convinced that the students who will join the group soon, Andrea, Andrew, and Thomas, will keep this great spirit alive. I am also grateful to the informal members of the group, who visited us. Thank you, Jack, for trying to bring more Isabelle to Viper, and for somehow being very funny; Trayan, for the very fun collaboration on *Hyper Separation Logic* (even if you want to change the state model every other month).

Many former members of the group also shaped my PhD experience. Thank you, Alex, for being a great collaborator, a very insightful critic of my work, and for always trying to get to the essence of things; Michael, for being an amazing Viper Roots member, for pushing for these end-to-end theorems, and for showing me what being efficient *really* means; Wolf, for your incredible wisdom and fascinating (but wrong) takes;

Caterina, for being such a great person to hang out with at conferences (and do research at the *research tables*); Malte, for regularly joining Super Kondi and for co-supervising with me; Alexandra, for the many encouragements and conversations about academia and life; Xavier, for bringing French cheese culture to the group; Arshavir, for *almost* being my office mate; Federico, for having been my COOP (briefly) and Program Verification TA; Jérôme, for sharing with me some insight on invariant inference that led to Chapter 3; Vytautas, for the *healthy* competition about the group food and for making me appreciate the great nutritional value of soup; Christoph, for helping me with a draft on inlining and for an early discussion on Hyper Hoare Logic; Martin, for helping me notice some unsoundnesses in Viper; Wytse, for bringing VerCors knowledge to the group; and Fábio, for fixing so many GitHub issues.

I additionally want to thank the new group in town for the fun lunch breaks, activities, and interesting discussions. Thank you, Ralf, for being my second advisor (now you cannot say that you didn't know); Isaac, for getting COVID with me after OPLSS in Boston; Johannes, for always being honest; Max, for the fun parties at your place and in Hawaii; and Rudy, for being a big fan of the unofficial Viper Roots logo.

In the summer of 2023, I had the privilege to intern at Microsoft Research (MSR) in Redmond in the RiSE team. Thank you to my collaborators Guido, Nik, Tahina, Aseem, Megan, and Gabriel, for the interesting work we did together. It was fascinating to see how research is done in an industrial lab, and to learn more about F*. Thank you to other RiSE members, in particular to Margus for the boat ride, Nikolaj for the wine, and Galen for the flight. Thank you to my fellow interns and hotelmates who made this an incredible summer; I'm very happy we won this *football* game!

A particularly rewarding part of the PhD was supervising students. I am grateful to have had the opportunity to supervise so many talented students: Yanick Bachmann, Benjamin Bonneau, Noé Weeks, Matthias Roshardt, Nicola Widmer, Matthias Schenk, Dina Weiersmüller, Anqi Li, Ellen Arlt, Hongyi Ling, Yiqun Liu, Yushuo Xiao, Trayan Gospodinov, Paul Winkler, Patrick Neugebauer, Ramon Wick, and David Hagen.

Outside of research, my time in Zürich was made immensely richer by friends that I made along the way (they say it's the real treasure after all). When I first came to Zürich in 2018 for my Master's studies, I was accompanied by many friends from Polytechnique, with whom I had the pleasure to share many so-called "French-speaking dinners". Thank you, Elías, for having always been there for me, be it in France, in Spain, in Switzerland, or in the US, for sharing with me your hard earned wisdom about life, and for insisting on the wine. You still owe me a position in your future government! Thank you, Raph and Anouk, for having been with me from Lyon, to Paris, and then to Zürich, and for all the good times in between; Thomas, for all the hikes, the football and tennis games, the California adventure, and for the amazing trips to Cambodia and Greece; Jean-Charles, for all the football games (played and watched) and for *really* trying to get the DJ to play *ramener la coupe à la maison* (I heard him say it's the next one!); and Lucien, for our political lunches to save the motherland.

I was also fortunate to leave my bubble and make many international friends during my time in Zürich, who equally shaped my experience, starting with my many amazing flatmates from Bächlerstrasse. In particular, thank you, Maggie, Kai, and Wang Li, for the amazing trip to Vietnam, and Kathleen-Jeanne, for the great moments in Seattle. I was also lucky to meet many amazing people through my PhD. Thank you, Anu, for inviting me to give a talk at the ZISC seminar and then not showing up for it, for *almost* inviting me to Indonesia, and for your *special* British humor; Irina, for collecting *those* quotes, for being such a fan of NYU, and for the paper you wrote about me (actually not sure about this last one); Sofia, for all those Tiny Fish lunches, for organizing these amazing Swiss Sunday hikes, and for helping me practice my French; Foteini, for teaching me how computers *actually* work and for the amazing visits to Portland and Mount Rainier; Martin and Charlyne, for *sometimes* inviting me to your place, and for the amazing kitesurfing trip to Cape Verde. One thing that I will dearly miss from ETH, especially on Tuesday at 6pm (CEST), is *Super Kondi* (a mix of cardio, strength, clubbing, with cult vibes (but in a good way)) and the Super Kondi crew (aka the InfSec Institute): Friederike, Matilda, Felix, Simon, Srdjan, Francesca, Daniele, Varun, Matteo, and many others. Thank you also to Colin, Daria, Mélisande, and many others for making my time in Zürich so much fun.

This PhD journey began long before ETH, shaped by many earlier mentors. During my early undergrad years at Lycée du Parc, Franz Ridde taught me how to do mathematics *properly*, while Marc Pauly taught me how

to see the beauty in mathematics. I then went to École Polytechnique, where Stéphane Graham-Lengrand and Jean-Christophe Filliâtre made me discover the beauty behind programming languages, while Benjamin Doerr introduced me to research. Thank you, Gautier, for taking the initiative to start our first ever research project and inviting Raph and me to join, showing me what being an adult looks like, and for all the fun moments. I then pursued research at SRI International, where I learned a lot from Susmit Jha and Natarajan Shankar. I finally joined ETH, where Dmitriy Traytel and Christoph Sprenger introduced me to Isabelle/HOL. Thank you everyone for having contributed to my growth as a researcher.

I owe more to my family than I could ever put into words. To my sisters, Laurène and Tiphaine, thank you for always believing in me; you've been my *fearless* supporters all along. To my parents, Sylvie and Bernard, thank you for giving me the freedom to follow my own path, for trusting me even when it led far from home, and most importantly for teaching me how to be happy. I could not have done this without your support, encouragement, and love. You have shaped me into the person I am today, and I hope I have made you proud.

Last but not least, thank you, Caro, for everything. Thank you for our silly moments, our deep conversations, our amazing trips (and it's only the start!), and for trying to *fix* my English. I met you at a time when I felt lost in life, and you have given me so much clarity, happiness, and hope since then. I very much look forward to the rest of our life, be it in Switzerland, in New York (MRC!), or anywhere else in the world. As they (all) say, *the future look goods*.

Zürich, September 2025

Contents

List of Figures

List of Tables

List of Examples

List of Definitions

List of Formal Results

1. Introduction

It starts with
One thing, I don't know why
It doesn't even matter how hard you try
Keep that in mind, I designed this rhyme
To explain in due time, all I know

Linkin Park, *In the End*

Computers play a central role in nearly every aspect of modern life, including communication, transportation, healthcare, and finance. As a consequence, bugs and security vulnerabilities in software can have severe consequences, such as financial losses, data breaches, and even deaths. To mitigate such risks, a variety of techniques have been proposed, including unit testing [1], fuzzing [2], property-based testing [3], bounded model checking [4–6], or runtime verification [7]. While these approaches are very effective at showing the *presence* of bugs, they cannot guarantee their *absence*. In particular, they may miss corner cases that could be exploited by malicious attackers.

A fundamentally different approach, called *formal verification*, is to construct a *mathematical proof* that a program is correct. Unlike the previously-mentioned techniques, formal verification does not miss corner cases, and thus provides a very high level of confidence in the correctness of a program. Mathematical proofs about programs are typically constructed using *program logics* such as *Hoare logic* [8, 9] or *separation logic* [10, 11]. A program logic is a formal proof system with a set of inference rules to prove properties of programs in a compositional way. However, manually constructing such proofs requires a lot of effort and expertise.

Automated deductive verifiers (*automated verifiers* or *verifiers* in short), such as Boogie [12], Dafny [13], F* [14], VeriFast [15], Viper [16], or Why3 [17], attempt to solve these issues by automating the construction of the proof. As depicted in Figure 1.1, automated verifiers are tools that take as input a program and a specification (describing what the program should and should not do, *e.g.*, in the form of a precondition and a postcondition for an entry method), and try to construct a mathematical proof showing that *all possible executions* of the program satisfy the specification. However, since checking whether a program satisfies its specification is undecidable [18, 19], automated verifiers additionally take *hints* (such as loop invariants) as input to help guide the proof search. Automated verifiers either report a *success*, indicating that *all possible executions* of the program satisfy the specification, or a *failure* (typically accompanied by one or more error messages), which either means that the program violates the specification, or that the verifier was not able to construct a proof, for example because of insufficient hints.

Automated verifiers have shown practical impact in industry. For example, code verified with the F* verifier [14] at Microsoft has been deployed in Microsoft Azure [20], as well as in Mozilla Firefox and the Linux kernel [21–23]. The Gobra verifier [24] for Go, built on top of the Viper infrastructure [16], has been used to verify the correctness of the reference

[1]: Huizinga et al. (2007), *Automated Defect Prevention*

[2]: Miller et al. (1990), *An Empirical Study of the Reliability of UNIX Utilities*

[3]: Claessen et al. (2000), *QuickCheck*

[4]: Biere et al. (1999), *Symbolic Model Checking without BDDs*
[5]: Clarke et al. (2001), *Bounded Model Checking Using Satisfiability Solving*
[6]: Biere et al. (2003), *Bounded Model Checking*

[7]: Leucker et al. (2009), *A Brief Account of Runtime Verification*

[8]: Floyd (1967), *Assigning Meanings to Programs*
[9]: Hoare (1969), *An Axiomatic Basis for Computer Programming*

[10]: Reynolds (2002), *Separation Logic*
[11]: O'Hearn (2007), *Resources, Concurrency, and Local Reasoning*

[12]: Leino (2008), *This Is Boogie 2*

[13]: Leino (2010), *Dafny*

[14]: Swamy et al. (2016), *Dependent Types and Multi-Monadic Effects in F**

[15]: Jacobs et al. (2011), *VeriFast*

[16]: Müller et al. (2016), *Viper*

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

[18]: Turing (1937), *On Computable Numbers, with an Application to the Entscheidungsproblem*
[19]: Rice (1953), *Classes of Recursively Enumerable Sets and Their Decision Problems*
[14]: Swamy et al. (2016), *Dependent Types and Multi-Monadic Effects in F**

[20]: Ramananandro et al. (2019), *{EverParse}*

[21]: Zinzindohoué et al. (2017), *HACL**
[22]: Bond et al. (2017), *Vale*
[23]: Protzenko et al. (2020), *EverCrypt*

[24]: Wolf et al. (2021), *Gobra*

[16]: Müller et al. (2016), *Viper*

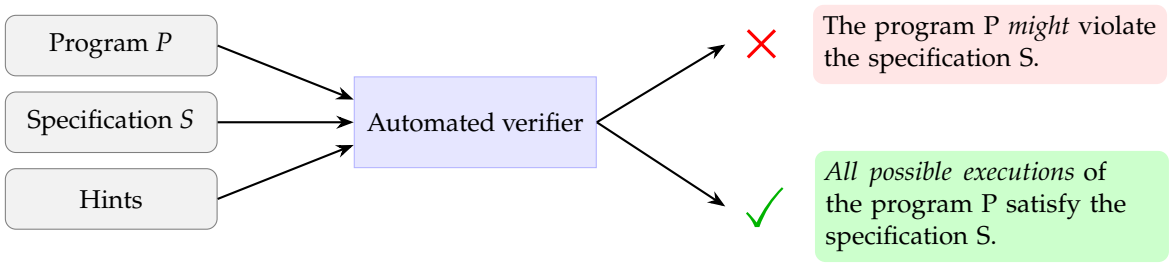

Figure 1.1.: High-level representation of an automated (deductive) verifier.

implementation of SCION's next-generation internet router [25]. The authorization engine at the core of Amazon Web Services, which runs a *billion times per second*, has been formally verified [26] with the Dafny verifier [13]. Despite these successes, modern verifiers face two key challenges: *trustworthiness* and *expressiveness*.[1]

[25]: Pereira et al. (2024), *Protocols to Code*

[26]: Chakarov et al. (2025), *Formally Verified Cloud-Scale Authorization*

[13]: Leino (2010), *Dafny*

1: Modern verifiers face other challenges, including automation, scalability, performance, debugging, etc. This thesis only addresses the two mentioned challenges.

Trustworthiness. To provide meaningful guarantees, automated verifiers must be *sound*, *i.e.*, they should report success *only* when the program actually satisfies the specification. However, modern automated verifiers often lack dedicated formal foundations that accurately capture their actual verification algorithms. Modern verifiers use complex techniques optimized for performance and automation, which are typically absent from existing theoretical frameworks. As a result, *unsoundnesses* (*i.e.*, when the verifier *incorrectly* reports that the program satisfies the specification) are discovered regularly, undermining the *trustworthiness* of verifiers.

Expressiveness. Automated verifiers are limited in their *expressiveness*. While they are capable of establishing correctness properties, they fall short when it comes to establishing *hyperproperties* [27], an important class of properties that relate multiple executions of a program, and that can express important properties such as (generalized) non-interference [28, 29], a security property ensuring that public outputs do not leak confidential data.

[27]: Clarkson et al. (2008), *Hyperproperties*

[28]: McCullough (1988), *Noninterference and the Composability of Security Properties*
[29]: Goguen et al. (1982), *Security Policies and Security Models*

This thesis addresses these two challenges. To address the *trustworthiness* challenge, we develop novel formal foundations to justify the soundness of existing automated verifiers based on *separation logic* [10, 11], such as Gillian [30], VeriFast [15], and Viper [16]. To address the *expressiveness* challenge, we develop a novel program logic (*Hyper Hoare Logic*) and a novel automated verifier (Hypra) for hyperproperties.

[10]: Reynolds (2002), *Separation Logic*
[11]: O'Hearn (2007), *Resources, Concurrency, and Local Reasoning*

[30]: Fragoso Santos et al. (2020), *Gillian, Part i*

[15]: Jacobs et al. (2011), *VeriFast*

[16]: Müller et al. (2016), *Viper*

1.1. Trustworthiness

The first objective of this thesis is to make automated verifiers based on *separation logic* more trustworthy. *Separation logic* (SL) [10, 11, 31] refers to a broad class of state-of-the-art modular[2] program logics for sequential and concurrent programs that manipulate mutable data structures on the heap.

[10]: Reynolds (2002), *Separation Logic*
[11]: O'Hearn (2007), *Resources, Concurrency, and Local Reasoning*
[31]: Jung et al. (2018), *Iris from the Ground Up*

2: Meaning that properties proven about each method and thread separately can be composed to prove properties about whole programs.

In the following, we present the state of the art for building *sound and automated* verifiers based on separation logic, and then we discuss the challenges that remain to be addressed to make modern SL-based automated verifiers more trustworthy.

1.1.1. State of the Art

On a high-level, we can distinguish two approaches to build *sound and automated* verifiers based on separation logic. The *soundness-first* approach starts with a program logic that has been formally proven sound in an *interactive theorem prover*, and then adds automation to the construction of a proof in this program logic, typically by developing dedicated tactics. While sound, this method lacks the high degree of automation achieved by modern automated verifiers, which are designed around SMT-based automation from the start. In contrast, the *automation-first* approach, which this thesis is about, starts with an existing SMT-based automated verifier that enjoys a high degree of automation, and establishes its soundness retroactively.

Soundness-first approach

Interactive theorem provers (ITPs), such as Rocq[3] [32], Isabelle/HOL [33], or Lean [34], are more trustworthy than automated verifiers, because every proof is checked by a small kernel that enforces the rules of a well-defined formal logic. Thus, one natural approach to build a *sound and automated* verifier is to first embed a program logic (such as separation logic [10, 11]) in an ITP, and then add automation in the form of tactics [35–40]. We call such verifiers *tactic-based verifiers*. For example, DiaFrame [38], an automated verifier for fine-grained concurrent programs, is implemented as a Rocq library on top of *Iris* [31, 41], a higher-order concurrent separation logic framework embedded in the Rocq prover. RefinedC [37] and RefinedRust [39], two automated verifiers for C and Rust programs, also build on Iris: Given a C or Rust program annotated with specifications and hints, they automatically generate proofs that are then checked by Rocq. VST-Floyd [36] and VST-A [40] add automation on top of VST's separation logic [42], which is embedded in Rocq, to verify C programs. Tactic-based verifiers are sound by construction[4], as each successful run results in a proof in an ITP, but they typically offer less automation than SMT-based verifiers.

Another way to obtain a *sound and automated* verifier is to implement the verifier in an ITP and then extract the implementation. For example, adaptations of SmallFoot [43], the first automated verifier based on symbolic execution for separation logic [44], have been implemented in HOL [45] and in Rocq [46]. More recently, Keuchel et al. [47] have developed Katamaran, a symbolic execution engine for instruction set architectures verified in Rocq.

Another example based on this approach is Steel [48], an SL-based proof-oriented programming language embedded in F*.[5] Steel programs are automatically proven correct using a type checker that is proved sound against SteelCore [49], a concurrent separation logic proven sound in F*;[6] the type checker uses the SMT solver Z3 [52] to discharge proof

3: Previously known as Coq.

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

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

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

[10]: Reynolds (2002), *Separation Logic*
[11]: O'Hearn (2007), *Resources, Concurrency, and Local Reasoning*

[35]: Chlipala (2011), *Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic*
[36]: Cao et al. (2018), *VST-Floyd*
[37]: Sammler et al. (2021), *RefinedC*
[38]: Mulder et al. (2022), *Diaframe*
[39]: Gäher et al. (2024), *RefinedRust*
[40]: Zhou et al. (2024), *VST-A*

[38]: Mulder et al. (2022), *Diaframe*

[31]: Jung et al. (2018), *Iris from the Ground Up*
[41]: Jung et al. (2015), *Iris*

[37]: Sammler et al. (2021), *RefinedC*

[39]: Gäher et al. (2024), *RefinedRust*

[36]: Cao et al. (2018), *VST-Floyd*

[40]: Zhou et al. (2024), *VST-A*

[42]: Appel (2011), *Verified Software Toolchain*

4: If the logic on which they are based is also sound.

[43]: Berdine et al. (2005), *Smallfoot*

[44]: Berdine et al. (2005), *Symbolic Execution with Separation Logic*

[45]: Tuerk (2009), *A Formalisation of Smallfoot in HOL*

[46]: Appel (2011), *VeriSmall*

[47]: Keuchel et al. (2022), *Verified Symbolic Execution with Kripke Specification Monads (and No Meta-Programming)*

[48]: Fromherz et al. (2021), *Steel*

5: While we primarily consider F* as an automated verifier, it can also be considered, to some extent, as an interactive theorem prover. Compared to classical ITPs, F* has a larger trusted code base, as it interacts with the SMT solver in non-trivial ways.

[49]: Swamy et al. (2020), *SteelCore*

6: It has recently been noted [50] that SteelCore's model relies on a non-standard axiom for monotonic state [51], which is unsound when combined with the strong excluded middle axiom.

[52]: de Moura et al. (2008), *Z3*

obligations.

Automation-first approach

In contrast to the soundness-first approach, many practical automated verifiers based on separation logic are not embedded in an ITP but are rather implemented in efficient mainstream programming languages, and they often include subtle optimizations and advanced automation that are absent from existing theoretical frameworks. Those include jStar [53], VCC [54], VeriFast [15], GRASShopper [55], Viper [16], Caper [56], VerCors [57], Nagini [58], Prusti [59], Gillian [30], Gobra [24], CN [60], or TPot [61]. Notably, several among them are *translational verifiers*, *i.e.*, they are organized into a *front-end*, which encodes an input program along with its specification and verification logic into an *intermediate verification language* (IVL), and a *back-end*, which computes proof obligations from the IVL program and discharges them, for instance, using an SMT solver such as Z3 [52] or CVC5 [62]. Developing a program verifier on top of an IVL has major engineering benefits. For example, back-end verifiers, which often contain complex proof search algorithms, sophisticated optimizations, and functionality to communicate with solvers and to report errors, can be re-used across different verifiers, which reduces the effort of developing a program verifier dramatically. Examples of such *translational verifiers* include Gillian [30] (for C, JavaScript [63] and Rust [64]), VeriFast [15] (for C, Java, C++ [65], and Rust [66]), and Viper [16] (with support for C, Java [57], Rust [59], Go [24], Python [58], and others).

Several works have attempted to provide formal foundations for practical automated verifiers based on separation logic to justify their soundness. Jacobs et al. [67] have formalized and proved sound, in Rocq, the soundness of a simplified version of the symbolic execution used in VeriFast [15]. There have also been formalizations, on paper, of the core approach taken by Gillian [30]: Maksimović et al. [68] briefly describe a parametric soundness framework and show soundness of the resulting symbolic execution, while Lööw et al. [69] present a formal compositional symbolic execution engine inspired by Gillian. Smans et al. [70] introduce *implicit dynamic frames*, a variant of separation logic on which Viper [16] and its front-end verifiers are based, and prove, on paper, the soundness of a verification conditions generator for a small imperative language. Zimmerman et al. [71] formalize and prove sound, also on paper, a simplified variant of Viper's symbolic execution back-end verifier targeted at gradual verification [72]. In work not presented in this thesis [73], we give a semantics to a high-level parametric verification language that captures the essence of verifiers such as GRASShopper [55], VeriFast, and Viper, for the purpose of showing formal results on method call inlining and loop unrolling. In another work not presented in this thesis [74] (and led by Gaurav Parthasarathy), we formalize a low-level semantics for a subset of Viper in Isabelle/HOL and instrument one of Viper's back-end verifiers to generate per-run certificates of correctness in Isabelle/HOL.

[53]: Distefano et al. (2008), *jStar*

[54]: Cohen et al. (2009), *VCC*

[15]: Jacobs et al. (2011), *VeriFast*

[55]: Piskac et al. (2014), *GRASShopper*

[16]: Müller et al. (2016), *Viper*

[56]: Dinsdale-Young et al. (2017), *Caper*

[57]: Blom et al. (2017), *The VerCors Tool Set*

[58]: Eilers et al. (2018), *Nagini*

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

[30]: Fragoso Santos et al. (2020), *Gillian, Part i*

[24]: Wolf et al. (2021), *Gobra*

[60]: Pulte et al. (2023), *CN*

[61]: Cebeci et al. (2024), *Practical Verification of System-Software Components Written in Standard C*

[52]: de Moura et al. (2008), *Z3*

[62]: Barbosa et al. (2022), *Cvc5*

[30]: Fragoso Santos et al. (2020), *Gillian, Part i*

[63]: Maksimović et al. (2021), *Gillian, Part II*

[64]: Ayoun et al. (2025), *A Hybrid Approach to Semi-automated Rust Verification*

[15]: Jacobs et al. (2011), *VeriFast*

[65]: Mommen et al. (2022), *Verification of C++ Programs with VeriFast*

[66]: Foroushaani et al. (2022), *Modular Formal Verification of Rust Programs with Unsafe Blocks*

[16]: Müller et al. (2016), *Viper*

[57]: Blom et al. (2017), *The VerCors Tool Set*

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

[24]: Wolf et al. (2021), *Gobra*

[58]: Eilers et al. (2018), *Nagini*

[67]: Jacobs et al. (2015), *Featherweight VeriFast*

[15]: Jacobs et al. (2011), *VeriFast*

[30]: Fragoso Santos et al. (2020), *Gillian, Part i*

[68]: Maksimović et al. (2021), *Gillian*

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

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

[16]: Müller et al. (2016), *Viper*

[71]: Zimmerman et al. (2024), *Sound Gradual Verification with Symbolic Execution*

[72]: Bader et al. (2018), *Gradual Program Verification*

[73]: Dardinier et al. (2023), *Verification-Preserving Inlining in Automatic Separation Logic Verifiers*

1.1.2. Challenges

Remarkably, despite substantial progress in the literature, a significant gap remains between existing theoretical foundations and the practical automated verifiers based on separation logic that employ advanced automation, as we explain next.

[55]: Piskac et al. (2014), *GRASShopper*

[74]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers*

Gap 1: Translational verifiers

As explained earlier, several modern SL-based verifiers are *translational verifiers*, such as Gillian [30], VeriFast [15], and Viper [16]. Formally reasoning about translational verifiers, in particular proving their soundness, is more difficult than for verifiers developed by embedding a program logic in an interactive theorem prover. Indeed, proving that a translational verifier is sound requires (1) a formal semantics of the IVL as well as proofs that connect the IVL program (2) to the verification back-end and (3) to the input program. While these steps have been studied for IVLs based on standard first-order logic [75–77], they pose additional challenges for IVLs that natively support more-complex widely-used reasoning principles such as those of separation logic (and variations such as implicit dynamic frames [70]).

[30]: Fragoso Santos et al. (2020), *Gillian, Part i*

[15]: Jacobs et al. (2011), *VeriFast*

[16]: Müller et al. (2016), *Viper*

[75]: Parthasarathy et al. (2021), *Formally Validating a Practical Verification Condition Generator*

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

[77]: Herms (2013), *Certification of a Tool Chain for Deductive Program Verification*

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

Gap 2: Advanced separation logic features

Additionally, modern SL-based verifiers employ advanced Separation Logic (SL) features that are not covered by existing theory.

For example, several automated verifiers [24, 57, 58] support *magic wands* (also called *separating implication*), an important SL connective, which is in particular useful to specify properties of partial data structures, for instance during iterative traversals of lists or trees. Surprisingly, as we discovered in this thesis, existing support for magic wands in automated verifiers was either manual [78] or unsound [79], illustrating the need for dedicated mechanized formal foundations.

As another example, many modern verifiers [15, 16, 24, 57, 58, 80] support *fractional predicates*, a generalization of *fractional permissions* [81, 82] to arbitrary SL predicates such as (co)inductive predicates and magic wands, important for example to reason about concurrent reads of shared data structures. Surprisingly, as we also discovered in this thesis, there is a fundamental discrepancy between existing theory about fractional predicates [83, 84] and the rules used by these verifiers.

Yet another gap comes from the fact that automated verifiers based on Viper [16, 24, 57–59] combine several advanced features and permission models from separation logic [10], which are typically defined using separation algebras [31, 85, 86], with *implicit dynamic frames* [70], where states are typically formalized as total heaps [87].

[24]: Wolf et al. (2021), *Gobra*

[57]: Blom et al. (2017), *The VerCors Tool Set*

[58]: Eilers et al. (2018), *Nagini*

[78]: Blom et al. (2015), *Witnessing the Elimination of Magic Wands*

[79]: Schwerhoff et al. (2015), *Lightweight Support for Magic Wands in an Automatic Verifier*

[15]: Jacobs et al. (2011), *VeriFast*

[16]: Müller et al. (2016), *Viper*

[24]: Wolf et al. (2021), *Gobra*

[57]: Blom et al. (2017), *The VerCors Tool Set*

[58]: Eilers et al. (2018), *Nagini*

[80]: Leino et al. (2009), *Verification of Concurrent Programs with Chalice*

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

[82]: Bornat et al. (2005), *Permission Accounting in Separation Logic*

[83]: Le et al. (2018), *Logical Reasoning for Disjoint Permissions*

[84]: Brotherston et al. (2020), *Reasoning over Permissions Regions in Concurrent Separation Logic*

[16]: Müller et al. (2016), *Viper*

[24]: Wolf et al. (2021), *Gobra*

[57]: Blom et al. (2017), *The VerCors Tool Set*

[58]: Eilers et al. (2018), *Nagini*

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

[10]: Reynolds (2002), *Separation Logic*

[31]: Jung et al. (2018), *Iris from the Ground Up*

[85]: Calcagno et al. (2007), *Local Action and Abstract Separation Logic*

[86]: Dockins et al. (2009), *A Fresh Look at Separation Algebras and Share Accounting*

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

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

Table 1.1.: (Non-exhaustive) list of hyperproperties, classified according to their (quantification) types. CRDTs stand for *conflict-free replicated data types* [97].

Type	Violation	Property	Example use case
$\forall$	$\exists$	Safety properties	Absence of runtime errors, functional correctness
$\exists$	$\forall$	Reachability	Bug finding [89], test input generators [90]
$\forall\forall$	$\exists\exists$	Non-interference	Information flow security
		Determinism	Cryptographic hash functions
		Monotonicity	CRDTs [91]
		Commutativity	Parallel aggregations (*e.g.*, MapReduce [92]), CRDTs
		Idempotence	CRDTs
		Function sensitivity	Differential privacy [93]
$\forall\forall\forall$	$\exists\exists\exists$	Transitivity	Custom comparators [88]
		Homomorphism	Homomorphic encryption [94]
$\exists\exists$	$\forall\forall$	Non-determinism	Random generator, test flakyness
$\forall\forall\forall\forall$	$\exists\exists\exists\exists$	Associativity	Parallel reductions (*e.g.*, MapReduce [92]), CRDTs
		Robust declassification [95]	Information flow security with declassification
$\forall\exists$	$\exists\forall$	Semantic parameter usage [96]	Dead-parameter elimination
$\forall\forall\exists$	$\exists\exists\forall$	Generalized non-interference [28]	Information flow security for non-deterministic programs
$\exists\forall$	$\forall\exists$	Existence of a minimum	Optimization algorithms

1.2. Expressiveness

The second objective of this thesis is to improve the expressiveness of automated verifiers by developing novel formal foundations for reasoning about a large class of *hyperproperties* [27] and a novel automated verifier based on these foundations. Many important security and functional properties are hyperproperties, *i.e.*, properties that relate multiple executions of a program. For example, *information flow security*, which ensures that confidential data does not leak through public outputs, can be formalized as *non-interference* [29]. Non-interference requires that any two executions with the same public inputs (but potentially different confidential inputs) have the same public outputs. Non-interference is an example of a 2-*safety hyperproperty*, *i.e.*, a property that should hold for all pairs of executions of a program. Another example of hyperproperty is *transitivity*, which custom comparators between objects (*e.g.*, in Java) must satisfy (*e.g.*, for sorting to be correct). Transitivity is a 3-*safety hyperproperty*, as it compares three executions of a program [88]: If two runs of a comparator `compare(a, b)` and `compare(b, c)` both return a positive result, then the third run `compare(a, c)` should also return a positive result. In this thesis, we often refer to *k-safety hyperproperties* as $\forall^k$*-properties*, as they can be expressed with k universal quantifiers over executions of a program, where k is the number of executions that the property relates. For example, non-interference is a $\forall\forall$*-property*, and transitivity is a $\forall\forall\forall$*-property*.

[27]: Clarkson et al. (2008), *Hyperproperties*

[29]: Goguen et al. (1982), *Security Policies and Security Models*

[88]: Sousa et al. (2016), *Cartesian Hoare Logic for Verifying K-Safety Properties*

Many interesting hyperproperties fall outside the class of $\forall^k$-properties, as shown in Table 1.1. For example, *reachability* is an $\exists$*-property*, as it requires the existence of at least one execution that satisfies a given property. Moreover, some hyperproperties require both universal and existential quantification over executions, such as *generalized non-interference* [28] ($\forall\forall\exists$), a weaker form of non-interference suitable for non-deterministic programs or the existence of an execution with minimal output values ($\exists\forall$). Finally, *violations* of $\forall^k$-properties also fall outside the class of $\forall^k$-properties: To prove a violation of a $\forall^k$-property, one must show the existence of k executions that together violate the property, which

[28]: McCullough (1988), *Noninterference and the Composability of Security Properties*

Table 1.2.: (Non-exhaustive) overview of program logics for hyperproperties, classified in two dimensions. The type of properties a logic can establish, and the number of program executions these properties can relate (column "∞" subsumes an unbounded and an infinite number of executions). The distinction between $\exists^*$ (backward) and $\exists^*$ (forward) is explained in Appendix A.3. A green checkmark indicates that a property is handled by Hyper Hoare Logic, a novel contribution of this thesis, and ∅ indicates that no other program logic supports it.

	Number of executions			
Type	**1**	**2**	***k***	**∞**
$\forall^*$ (hypersafety)	✓ HL [8, 9], OL [102], RHL [103], CHL [88], RHLE [96], MHRM [104], BiKAT [105]	✓ RHL [103], CHL [88], RHLE [96], MHRM [104], BiKAT [105]	✓ CHL [88], RHLE [96]	✓ ∅
$\exists^*$ (backward)	✓ IL [89, 106], InSec [107], BiKAT [105]	✓ InSec [107], BiKAT [105]	✓ ∅	✓ ∅
$\exists^*$ (forward)	✓ OL [102], RHLE [96], MHRM [104], BiKAT [105]	✓ RHLE [96], MHRM [104], BiKAT [105]	✓ RHLE [96]	✓ ∅
$\forall^*\exists^*$	not applicable	✓ RHLE [96], MHRM [104], BiKAT [105]	✓ RHLE [96]	✓ ∅
$\exists^*\forall^*$	not applicable	✓ ∅	✓ ∅	✓ ∅

corresponds to an $\exists^k$-property. Program logics for proving the existence of executions have been successfully used as foundation of industrial bug-finding tools [98–101].

Despite their importance, very few automated verifiers are able to establish hyperproperties. In the following, we present the state of the art for program logics (summarized in Table 1.2) and automated verifiers for hyperproperties, and then discuss the challenges that remain to be addressed to build an automated verifier that can be used to verify all hyperproperties presented in Table 1.1.

[98]: Blackshear et al. (2018), *RacerD*
[99]: Gorogiannis et al. (2019), *A True Positives Theorem for a Static Race Detector*
[100]: Distefano et al. (2019), *Scaling Static Analyses at Facebook*
[101]: Le et al. (2022), *Finding Real Bugs in Big Programs with Incorrectness Logic*

1.2.1. State of the Art

Program logics for hyperproperties

Deductive verification started with Floyd's seminal work on assigning meanings to programs [8], followed by *Hoare Logic* [9], a program logic designed to formally prove functional correctness of computer programs. Hoare Logic is widely used to prove the absence of runtime errors, functional correctness, resource bounds, etc. However, classical Hoare Logic cannot reason about hyperproperties, as it is limited to $\forall$-properties. To overcome such limitations and to reason about more types of properties, Hoare Logic has been extended and adapted in various ways.

Among them are several logics that can establish properties of two [103, 104, 108–115] or even k [88, 116, 117] executions of the same program, where $k > 2$ is useful for properties such as transitivity and associativity. *Relational logics* are able to prove *relational properties*, *i.e.*, properties relating executions of two (potentially different) programs, for instance, to prove program equivalence.

All of these logics have in common that they can prove only properties that hold *for all* (combinations of) executions ($\forall^k$-properties), that is, they prove the *absence* of bad (combinations of) executions; to achieve that, their judgments *overapproximate* the possible executions of a program. Overapproximate logics cannot prove the *existence* of (combinations of) executions and thus cannot establish certain interesting program properties, such as the presence of a bug or non-determinism.

To overcome this limitation, recent work [89, 106, 107, 118, 119] proposed program logics that can prove the *existence* of (individual) executions ($\exists$-properties), for instance, to *disprove* functional correctness. We call

[8]: Floyd (1967), *Assigning Meanings to Programs*
[9]: Hoare (1969), *An Axiomatic Basis for Computer Programming*
[103]: Benton (2004), *Simple Relational Correctness Proofs for Static Analyses and Program Transformations*
[104]: Maillard et al. (2019), *The next 700 Relational Program Logics*
[108]: Francez (1983), *Product Properties and Their Direct Verification*
[109]: Naumann (2020), *Thirty-Seven Years of Relational Hoare Logic*
[110]: Yang (2007), *Relational Separation Logic*
[111]: Aguirre et al. (2017), *A Relational Logic for Higher-Order Programs*
[112]: Amtoft et al. (2006), *A Logic for Information Flow in Object-Oriented Programs*
[113]: Costanzo et al. (2014), *A Separation Logic for Enforcing Declarative Information Flow Control Policies*
[114]: Ernst et al. (2019), *SecCSL*
[115]: Eilers et al. (2023), *CommCSL*
[88]: Sousa et al. (2016), *Cartesian Hoare Logic for Verifying K-Safety Properties*
[116]: D'Osualdo et al. (2022), *Proving Hypersafety Compositionally*
[117]: Gladshtein et al. (2024), *Mechanised Hypersafety Proofs about Structured Data*
[89]: O'Hearn (2019), *Incorrectness Logic*
[106]: de Vries et al. (2011), *Reverse Hoare Logic*
[107]: Murray (2020), *An Under-Approximate Relational Logic*
[118]: Raad et al. (2020), *Local Reasoning About the Presence of Bugs*
[119]: Raad et al. (2022), *Concurrent Incorrectness Separation Logic*

such program logics *underapproximate*. Tools based on underapproximate logics have proven useful for finding bugs on an industrial scale [98–101]. More recent work has proposed program logics that combine underapproximate and overapproximate reasoning for single executions (*i.e.*, $\forall$- and $\exists$-properties) such as Outcome Logic [102] and Exact Separation Logic [120], and for $\forall^*\exists^*$-properties, such as RHLE [96], BiKAT [105], and others [104, 121].

[98]: Blackshear et al. (2018), *RacerD*
[99]: Gorogiannis et al. (2019), *A True Positives Theorem for a Static Race Detector*
[100]: Distefano et al. (2019), *Scaling Static Analyses at Facebook*
[101]: Le et al. (2022), *Finding Real Bugs in Big Programs with Incorrectness Logic*
[102]: Zilberstein et al. (2023), *Outcome Logic*
[120]: Maksimović et al. (2023), *Exact Separation Logic*
[96]: Dickerson et al. (2022), *RHLE*
[105]: Antonopoulos et al. (2023), *An Algebra of Alignment for Relational Verification*
[104]: Maillard et al. (2019), *The next 700 Relational Program Logics*
[121]: Beutner (2024), *Automated Software Verification of Hyperliveness*

Automated verifiers for hyperproperties

Most existing automated verifiers (*e.g.*, Boogie [12], Dafny [13], Viper [16], Why3 [17]) are specifically designed to verify $\forall$-properties, *i.e.*, properties of *individual executions* of a program, such as functional correctness, absence of runtime errors, and resource bounds. While these verifiers are primarily designed for $\forall$-properties, they can be leveraged to verify $\forall^k$-properties (*k-safety hyperproperties*) by reducing the problem to standard safety verification. This is typically achieved via *self-composition* [122] or by constructing *product programs* [123, 124], which simulate multiple executions within a single program. Asymmetric product programs [125] can also be used to verify $\forall\exists$-properties.

[12]: Leino (2008), *This Is Boogie 2*
[13]: Leino (2010), *Dafny*
[16]: Müller et al. (2016), *Viper*
[17]: Filliâtre et al. (2013), *Why3 — Where Programs Meet Provers*
[122]: Barthe et al. (2004), *Secure Information Flow by Self-Composition*
[123]: Barthe et al. (2011), *Relational Verification Using Product Programs*
[124]: Eilers et al. (2019), *Modular Product Programs*
[125]: Barthe et al. (2013), *Beyond 2-Safety*

Deductive verifiers specifically designed for hyperproperties are mostly limited to $\forall^k$-properties. Examples include HyperViper (based on CommCSL) [115], SecC (based on SecCSL) [114], and WhyRel [126], for $\forall\forall$-properties, and Descartes (based on Cartesian Hoare Logic [88]) for $\forall^k$-properties.

[115]: Eilers et al. (2023), *CommCSL*
[114]: Ernst et al. (2019), *SecCSL*
[126]: Nagasamudram et al. (2023), *The WhyRel Prototype for Modular Relational Verification of Pointer Programs*
[88]: Sousa et al. (2016), *Cartesian Hoare Logic for Verifying K-Safety Properties*

Recently, several approaches [96, 121, 125, 127, 128] have been proposed to verify $\forall^*\exists^*$-properties. For example, ORHLE (based on RHLE) [96] and ForEx (based on FEHL) [121] are both based on program logics for $\forall^k\exists^l$-properties, where k and l must be fixed beforehand. It is, thus, not possible to compose proofs with different quantification schemes, *e.g.*, to use a $\forall\forall$-property and a $\forall\exists$-property in the same proof.

[96]: Dickerson et al. (2022), *RHLE*
[121]: Beutner (2024), *Automated Software Verification of Hyperliveness*
[125]: Barthe et al. (2013), *Beyond 2-Safety*
[127]: Unno et al. (2021), *Constraint-Based Relational Verification*
[128]: Beutner et al. (2022), *Software Verification of Hyperproperties Beyond K-Safety*

1.2.2. Challenges

Despite the existing literature, several challenges remain to build an automated verifier that can be used to verify all hyperproperties presented in Table 1.1.

Gap 1: Limited expressiveness of existing program logics

As shown by Table 1.2, existing program logics for hyperproperties face two main limitations. First, some types of hyperproperties cannot be expressed by any existing program logic (represented by $\varnothing$). For example, to prove that a program violates generalized non-interference [28], one needs to show that there *exist* two executions τ_1 and τ_2 such that *all* executions with the same high-sensitivity input as τ_1 have a different low-sensitivity output than τ_2.[7] Such $\exists^*\forall^*$-properties cannot be proved by any existing Hoare logic. Second, the existing logics cover different, often disjoint program properties, which may hinder practical applications: reasoning about a wide spectrum of properties of a given program

[96]: Dickerson et al. (2022), *RHLE*
[121]: Beutner (2024), *Automated Software Verification of Hyperliveness*

[28]: McCullough (1988), *Noninterference and the Composability of Security Properties*

7: Assuming no public (low-sensitivity) input.

requires the application of several logics, each with its own judgments; properties expressed in different, incompatible logics cannot be composed within the same proof system.

Gap 2: Automation

Even with a suitable program logic, there are many challenges to build a general-purpose SMT-based automated verifier for $\forall^*\exists^*$-properties and $\exists^*\forall^*$-properties. Such alternations of quantifiers are notoriously hard for SMT solvers (*e.g.*, to find witnesses for existentially-quantified executions), especially when different types of hyperproperties can interact in the same proof. Another key challenge is to design a verification algorithm to automatically verify loops while minimizing the amount of required user-provided hints (such as loop invariants or loop variants), as reasoning about loops in a relational setting is notoriously hard, especially when different executions perform a different number of iterations.

1.3. Contributions and Outline

In this thesis, we address the two aforementioned challenges: For the *trustworthiness* challenge, we provide novel formal foundations for existing automated verifiers based on separation logic, to justify their soundness. For the *expressiveness* challenge, we develop a new program logic (Hyper Hoare Logic) to reason about hyperproperties, along with a novel automated verifier (Hypra) based on this new program logic.

In the following, we describe the contributions of this thesis.

1.3.1. Trustworthiness: Formal Foundations for Verifiers based on Separation Logic

Contribution 1: Formal foundations for translational verifiers based on separation logic

Our first contribution, presented in Chapter 2, is a formal framework for proving the soundness of *translational verifiers* based on separation logic, such as Gillian [30], VeriFast [15], and Viper [16].

At the center of our framework is a generic intermediate verification language (IVL) called *CoreIVL*, which can be instantiated for a particular SL-based IVL. To support both Viper (based on implicit dynamic frames [70]) and Gillian and VeriFast (based on separation logic), CoreIVL is parametric over an *IDF algebra*, a novel generalization of separation algebras [85, 86] that capture both separation logic and implicit dynamic frames. We present two novel semantics for CoreIVL, which we prove equivalent: an operational semantics to connect to the back-end verifiers, and an axiomatic semantics to connect to the front-end program logic. Our operational semantics uses dual (*i.e.*, *angelic* and *demonic*) non-determinism to enable the application of different verification algorithms and heuristics in the back-end verifiers. Our axiomatic semantics

[30]: Fragoso Santos et al. (2020), *Gillian, Part i*

[15]: Jacobs et al. (2011), *VeriFast*

[16]: Müller et al. (2016), *Viper*

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

[85]: Calcagno et al. (2007), *Local Action and Abstract Separation Logic*

[86]: Dockins et al. (2009), *A Fresh Look at Separation Algebras and Share Accounting*

simplifies reasoning about the front-end translation by performing essential proof steps once and for all in the equivalence proof with the operational semantics, rather than for each concrete front-end translation. We demonstrate the practical utility of this framework by instantiating it with elements of the Viper IVL, and formally connecting to a front-end translation based on *concurrent separation logic* [11], as well as two Viper back-end verifiers: one based on *symbolic execution*, and one based on *verification condition generation* (which we have formally validated in work not presented in this thesis [74]).

[11]: O'Hearn (2007), *Resources, Concurrency, and Local Reasoning*

[74]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers*

Contribution 2: Formal foundations for fractional predicates

Our second contribution, presented in Chapter 3, is a novel formal semantics for SL assertions, which justifies the rules for (inductive and coinductive) *fractional predicates* used in practice by existing automated verifiers.

Fractional predicates are supported by existing SL verifiers such as VeriFast [15], Viper [16], VerCors [57], Nagini [58], and Gobra [24], via a notion of *syntactic multiplication*, which is *assumed* to satisfy three key properties: it distributes over assertions (*distributivity*), it permits fractions to be factored out from assertions (*factorizability*), and two fractions of the same assertion can be combined into one larger fraction (*combinability*). However, existing formal semantics [83, 84] for fractional predicates define multiplication *semantically* (via models), resulting in a semantics in which distributivity and combinability do not hold for key SL connectives such as magic wands, and fractions cannot be factored out from a separating conjunction. To resolve this discrepancy, we present a novel semantics for separation logic assertions allows states to hold more than a full permission to a heap location during the evaluation of an assertion. By reimposing upper bounds on the permissions held per location at statement boundaries, we retain key properties of (concurrent) separation logic, such as the frame rule and the parallel rule. Our novel assertion semantics, which enjoys distributivity, factorizability, and combinability, justifies the rules for (inductive and coinductive) fractional predicates used in practice by existing automated verifiers and can be used to extend this support to additional SL connectives, such as magic wands.

[15]: Jacobs et al. (2011), *VeriFast*

[16]: Müller et al. (2016), *Viper*

[57]: Blom et al. (2017), *The VerCors Tool Set*

[58]: Eilers et al. (2018), *Nagini*

[24]: Wolf et al. (2021), *Gobra*

[83]: Le et al. (2018), *Logical Reasoning for Disjoint Permissions*

[84]: Brotherston et al. (2020), *Reasoning over Permissions Regions in Concurrent Separation Logic*

Contribution 3: Formal foundations for automating magic wands

Our third contribution, presented in Chapter 4, is a formal framework to prove the soundness of *package algorithms* (*i.e.*, verification algorithms that automatically extract the resources required to satisfy a *magic wand* from a SL state), and the first sound and automated package algorithm for magic wands (built on this formal framework).

The key challenge of proving a magic wand $A \mathrel{-\!\!*} B$ (also called *packaging a wand*) is to find a *footprint*, *i.e.*, a state that, combined with any state in which A holds, yields a state in which B holds. Existing package algorithms (in Gobra, VerCors, Viper) either have a high annotation overhead or, as we show in this chapter, are unsound. We present a formal framework, which we call *package logic*, that precisely characterizes a wide

design space of possible package algorithms applicable to a large class of separation logics (as it is parametric over an IDF algebra). We prove that our package logic is sound and complete and use it to develop a novel package algorithm that offers competitive automation and, crucially, is sound. We also show that our package logic can be used with alternative definitions of magic wands, which we illustrate with a novel definition for magic wands that ensures that magic wands are combinable. We have implemented our techniques for Viper and demonstrate that they are effective in practice.

1.3.2. Expressiveness: A Novel Foundation and Verifier for Hyperproperties

Contribution 4: Hyper Hoare Logic, a novel program logic for hyperproperties

Our fourth contribution, presented in Chapter 5, is *Hyper Hoare Logic*, a novel program logic for hyperproperties, which supports arbitrary *program hyperproperties* over the terminating executions of a program, including $\forall^*\exists^*$- and $\exists^*\forall^*$-properties.

Hyper Hoare Logic (HHL) generalizes Hoare logic by lifting assertions from predicates over *individual* states to predicates over *sets* of states: It establishes *hyper-triples* of the form $[P]\ C\ [Q]$, where C is a program statement, and P and Q are *hyper-assertions* (*i.e.*, predicates over sets of states). In particular, we show that hyper-triples can express arbitrary hyperproperties over the terminating executions of a program, including hyperproperties that no other existing program logic can express (such as $\exists^*\forall^*$-properties), and we show how judgments of existing logics can be expressed as hyper-triples. To establish hyper-triples, we first present a minimal set of *core rules*, which are sound and complete (*i.e.*, every valid triple can be proven using the core rules). We then present a language for *syntactic hyper-assertions* (which restricts the interaction with the set of states to universal and existential quantification over states) and corresponding *syntactic rules* for atomic statements (such as assignments), which are easier to use than the core rules. We also present novel *loop rules*, which capture important reasoning principles. Finally, we present *compositionality rules*, which enable the flexible composition of hyper-triples of different forms and, thus, facilitate modular proofs.

Contribution 5: Hypra, a novel deductive verifier for hyperproperties based on Hyper Hoare Logic

Our fifth and last contribution, presented in Chapter 6, is Hypra, a novel automated deductive verifier for hyperproperties based on Hyper Hoare Logic (HHL), which can be used to prove $\forall^*\exists^*$- and $\exists^*\forall^*$-properties, as well as hyperproperties about runtime errors.

Hypra is a Viper front-end: It translates an input program and corresponding HHL specification and hints (such as loop invariants) into a Viper program and uses both Viper back-ends to verify the resulting program. Our key insight is that verification conditions for HHL can be encoded into a standard IVL by representing sets of states of the input

program explicitly in the states of the intermediate program. To support reasoning about runtime errors, we extend the definition of hyper-triples accordingly and add universal and existential quantifier over error states to the syntactic language of hyper-assertions. To avoid matching loops, where the underlying SMT solver gets stuck in an infinite instantiation of quantifiers, our encoding for loop-free statements separately tracks an upper bound and a lower bound of the set of reachable states at each program point. To verify loops, we show how to automatically select the right loop rule depending on the situation, and we present a novel loop rule that is more suitable for automated verification than the corresponding HHL rule. Finally, our evaluation on a set of benchmarks from the literature shows that Hypra can prove a large class of hyperproperties for a large class of programs, in a reasonable amount of time and with a reasonable amount of proof annotations.

1.4. Mechanization

All formal results presented in this thesis (definitions, lemmas, propositions, theorems) have been formalized in the interactive proof assistant Isabelle/HOL [33]. While publications based on each chapter have their own formalizations and artifacts available online [129–137], we have unified these formalizations into a single artifact for the thesis [138].

This artifact makes novel contributions over the ones associated with the individual publications, by increasing the compatibility between the individual results in the following ways. In the first part, the formalization of Chapter 4 is now based on the IDF algebra from Chapter 2, and the formalization of Chapter 3 is now based on a building block (*partial commutative monoid*) of the IDF algebra. Moreover, to show that the unbounded logic from Chapter 3 is compatible with important SL rules such as the parallel and the frame rule, we have additionally developed and proved sound a novel concurrent separation logic (CSL) based on a concrete unbounded model for SL states. We have shown that this concrete state model satisfies all the axioms described in Section 3.2.2, and that it is an instance of the IDF algebra from Chapter 2. Additionally, this unbounded CSL is based on the concurrent programming language defined in Chapter 2 (Section 2.5). Finally, in the second part, the formalization of the formal results from Chapter 6 is based on the formalization of Chapter 5.

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

[129]: Dardinier et al. (2024), *Formal Foundations for Translational Separation Logic Verifiers – Artifact*

[130]: Dardinier (2022), *Unbounded Separation Logic*

[131]: Dardinier et al. (2022), *Fractional Resources in Unbounded Separation Logic (Artifact)*

[132]: Dardinier (2022), *Formalization of a Framework for the Sound Automation of Magic Wands*

[133]: Dardinier (2022), *A Restricted Definition of the Magic Wand to Soundly Combine Fractions of a Wand*

[134]: Dardinier et al. (2022), *Sound Automation of Magic Wands (Artifact)*

[135]: Dardinier (2023), *Formalization of Hyper Hoare Logic: A Logic to (Dis-)Prove Program Hyperproperties*

[136]: Dardinier et al. (2024), *Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties (Artifact)*

[137]: Dardinier et al. (2024), *Hypra: A Deductive Program Verifier for Hyperproperties (Artifact)*

[138]: Dardinier (2025), *Artifact for the PhD Thesis "Formal Foundations for Automated Deductive Verifiers" (2025)*

1.5. Publications and Collaborations

The main results of this thesis have been presented in various publications.

Chapter 2 has been presented in

Formal Foundations for Translational Separation Logic Verifiers
Thibault Dardinier, Michael Sammler, Gaurav Parthasarathy, Alexander J. Summers, and Peter Müller
Proceedings of the ACM on Programming Languages (POPL) 2025 [139]

Chapter 3 has been presented in

Fractional Resources in Unbounded Separation Logic
Thibault Dardinier, Peter Müller, Alexander J. Summers
Proceedings of the ACM on Programming Languages (OOPSLA) 2022 [140]

Chapter 4 has been presented in

Sound Automation of Magic Wands
Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, Alexander J. Summers
International Conference on Computer Aided Verification (CAV) 2022 [141]

Chapter 5 has been presented in

Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties
Thibault Dardinier, Peter Müller
Proceedings of the ACM on Programming Languages (PLDI) 2024 [142]

Chapter 6 has been presented in

Hypra: A Deductive Program Verifier for Hyper Hoare Logic
Thibault Dardinier, Anqi Li, Peter Müller
Proceedings of the ACM on Programming Languages (OOPSLA) 2024 [143]

This thesis benefited from the contributions of several collaborators, in addition to my supervisor. In Chapter 2, Michael Sammler proved the soundness of both the symbolic execution (which he formalized) and the existing verification condition generation [74] with respect to ViperCore's operational semantics (Section 2.4). In Chapter 4, Gaurav Parthasarathy performed the evaluation described in Section 4.5, and Noé Weeks came up with the novel definition of combinable wands presented in Section 4.4.2 (as part of a summer internship at ETH Zurich cosupervised by Gaurav Parthasarathy and me[8]). Additionally, Alex Summers and Gaurav Parthasarathy contributed to the work on Chapter 2, Chapter 3, and Chapter 4, through numerous technical and non-technical discussions. Finally, Hypra, presented in Chapter 6, was implemented and evaluated by Anqi Li as part of her Master's thesis [144], which I supervised.

[74]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers*

8: In this section, I use the first person singular to distinguish my contributions in contrast to those of my collaborators.

[144]: Li (2023), *An Automatic Program Verifier for Hyperproperties*

Contributions beyond this thesis

In the time working on this thesis, I also contributed to the following publications, which are not part of this thesis.

A Formally Verified, Optimized Monitor for Metric First-Order Dynamic Logic
David Basin, Thibault Dardinier, Lukas Heimes, Srđan Krstić, Martin Raszyk, Joshua Schneider, Dmitriy Traytel
International Joint Conference on Automated Reasoning (IJCAR) 2020 [145]

Verification-Preserving Inlining in Automatic Separation Logic Verifiers
Thibault Dardinier, Gaurav Parthasarathy, Peter Müller
ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) 2023 [73]

CommCSL: Proving Information Flow Security for Concurrent Programs using Abstract Commutativity
Marco Eilers, Thibault Dardinier, Peter Müller
ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI) 2023 [115]

Towards Trustworthy Automated Program Verifiers: Formally Validating Translations into an Intermediate Verification Language
Gaurav Parthasarathy, Thibault Dardinier, Benjamin Bonneau, Peter Müller, Alexander J. Summers
ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI) 2024 [74]

PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs
Gabriel Ebner, Guido Martínez, Aseem Rastogi, Thibault Dardinier, Megan Frisella, Tahina Ramananandro, Nikhil Swamy
ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI) 2025 [50]

Automated Verifiers Based on Separation Logic

2. Translational Verifiers

Beware of bugs in the above code; I have only proved it correct, not tried it.

Donald Knuth

2.1. Introduction

As explained in Chapter 1, many modern automated verifiers are *translational verifiers*, *i.e.*, they are composed of a *front-end translation* of an input program with its specification (and hints to guide the proof search) into an *intermediate verification language* (IVL), and a *back-end verifier* that automates the verification of the IVL program. Examples of *translational verifiers* include Civl [146] and Dafny [13] based on the Boogie IVL [12], Creusot [147] and Frama-C [148] based on Why3 [17], Gillian for C and JavaScript [63] and Rust [64] based on GIL [30], as well as Prusti [59] and VerCors [57] based on Viper [16].

Developing a program verifier on top of an IVL has major engineering benefits. Most importantly, back-end verifiers, which often contain complex proof search algorithms, sophisticated optimizations, and functionality to communicate with solvers and to report errors, can be re-used across different verifiers, which reduces the effort of developing a program verifier dramatically.

On the other hand, formal reasoning about translational verifiers, in particular, proving their soundness, is more difficult than for verifiers developed by embedding a program logic in an interactive theorem prover (such as Bedrock [35], VST [36], and RefinedC [37]). Proving that a translational verifier is sound requires (1) a formal semantics of the IVL as well as proofs that connect the IVL program (2) to the verification backend and (3) to the input program. While these steps have been studied for IVLs based on standard first-order logic [75–77], they pose additional challenges for IVLs that natively support more-complex widely-used reasoning principles such as those of separation logic (SL) [10] (and variations such as implicit dynamic frames (IDF) [70]). We focus on these IVLs, which are commonly-used and especially useful for building verifiers for heap-manipulating and concurrent programs.

Challenge 1: Defining the semantics of the IVL. Standard programming languages and the intermediate languages used in compilers come with a notion of execution that can naturally be captured by an operational semantics. In contrast, IVLs are typically not designed to be executable, but instead to capture a wide range of verification problems and algorithms for solving them.

To capture different verification *problems*, IVLs contain features that enable the encoding of a diverse set of input programs (*e.g.*, by offering generic operations suitable for encoding different concurrency primitives),

[146]: Kragl et al. (2021), *The Civl Verifier*

[13]: Leino (2010), *Dafny*

[12]: Leino (2008), *This Is Boogie 2*

[147]: Denis et al. (2022), *Creusot*

[148]: Kirchner et al. (2015), *Frama-C*

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

[63]: Maksimović et al. (2021), *Gillian, Part II*

[64]: Ayoun et al. (2025), *A Hybrid Approach to Semi-automated Rust Verification*

[30]: Fragoso Santos et al. (2020), *Gillian, Part i*

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

[57]: Blom et al. (2017), *The VerCors Tool Set*

[16]: Müller et al. (2016), *Viper*

[35]: Chlipala (2011), *Mostly-Automated Verification of Low-Level Programs in Computational Separation Logic*

[36]: Cao et al. (2018), *VST-Floyd*

[37]: Sammler et al. (2021), *RefinedC*

[75]: Parthasarathy et al. (2021), *Formally Validating a Practical Verification Condition Generator*

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

[77]: Herms (2013), *Certification of a Tool Chain for Deductive Program Verification*

[10]: Reynolds (2002), *Separation Logic*

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

specifications (*e.g.*, by offering rich assertion languages), and verification logics (*e.g.*, by supporting concepts such as framing). An IVL semantics must reflect this generality. For instance, separation logic-based IVLs provide complex primitives for manipulating separation logic resources, which can be used to encode separation logic rules into the IVL. As a result, these primitives can be used to encode a large variety of input program features including procedure calls, loops, and concurrency.

To capture different verification *algorithms*, an IVL semantics must not prescribe *how* to construct a proof and should instead abstract over different algorithms. Back-ends should have the freedom to apply various techniques to compute proof obligations (*e.g.*, symbolic execution or verification condition generation), to resolve trade-offs between completeness and automation (*e.g.*, by over-approximating proof obligations), and to discharge proof obligations (*e.g.*, instantiating existentially quantified variables in different ways). For instance, existing algorithms have different performance characteristics for different classes of verification problems [149]; an IVL semantics should provide the freedom to choose the best one for the problem at hand. In practice, capturing different verification algorithms is important for verifiers with multiple back-ends for the same language (*e.g.*, based on either symbolic execution or verification condition generation). However, even a single back-end may offer a variety of different algorithms, which are chosen based on heuristics or configured by the user (*e.g.*, via command-line options or dedicated hints). For example, as we will see in Chapter 4, there is a wide design space of possible algorithms for automatically introducing a *magic wand*; an IVL semantics should not prescribe a specific algorithm, but instead allow the back-end to employ *any correct* algorithm. Moreover, back-ends along with their verification algorithms apply different algorithms over time, as their developers optimize existing verification algorithms or add support for new verification algorithms.

[149]: Eilers et al. (2024), *Verification Algorithms for Automated Separation Logic Verifiers*

Challenge 2: Connecting the IVL to back-ends. Soundness requires that successful verification of an IVL program by a back-end verifier implies the correctness of the IVL program. Since a back-end verifier's algorithm ultimately decides the outcome of a verification run, a soundness proof needs to formally connect the concrete verification algorithm to the IVL's semantics. In particular, this soundness proof needs to consider the proof search algorithms and optimizations performed by a concrete verification back-end and show that they produce correct results according to the IVL semantics. However, different back-ends typically use a diverse range of strategies to (for example) represent the program state, unroll recursive definitions, choose existentially-quantified permission amounts, and select the footprints of magic wands (as we will see in Chapter 4).

Challenge 3: Connecting the IVL to front-ends. Soundness also requires that the correctness of the IVL program implies the correctness of the *input* program with respect to its intended verification logic. Such soundness proofs are difficult due to the large semantic gap between input and IVL programs. The two programs may use different reasoning concepts and proof rules, which need to be connected by a soundness proof. This gap is particularly large for typical encodings into IVLs based

on separation logic, because the verification logic for the source of this translation is typically different from the one for the IVL program, e.g., one of the vast wealth of concurrent separation logics. For instance, a parallel composition of two threads in the input program is typically encoded as *three sequential* IVL programs: two for the parallel branches, each of which is verified using a separate specification provided by the user, and one for the enclosing code, which composes the two specifications to encode the behavior of the parallel composition overall. Such a translation of front-end proof rules into multiple sequential verification problems is not obvious; a soundness proof must bridge this gap.

Prior work. Several works formalize aspects of translational verifiers with IVLs based on separation logic, but none of them addresses all three challenges outlined above. For Viper, in work not presented in this thesis [74], we build a proof-producing version of Viper's verification condition generation back-end, but do not attempt to connect it to front-end languages nor give a general semantics for Viper that would also capture Viper's symbolic execution back-end. Similarly, Zimmerman et al. [71] formalize a version (only) of Viper's *symbolic execution* back-end; their focus is on adapting it to gradual verification. Jacobs et al. [67] show the soundness of the symbolic execution of VeriFast [15] w.r.t. an input C program.[1] However, VeriFast has only a single (symbolic execution) back-end that is used as the basis for multiple front-end languages (C, Java, Rust) and thus the formalization does not abstract over different verification algorithms.

[74]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers*

[71]: Zimmerman et al. (2024), *Sound Gradual Verification with Symbolic Execution*

[67]: Jacobs et al. (2015), *Featherweight VeriFast*

[15]: Jacobs et al. (2011), *VeriFast*

1: VeriFast itself is not an IVL, but must address similar challenges to IVLs based on separation logic since VeriFast's symbolic execution is used to justify multiple front-end languages (C, Java, Rust) using separation logic reasoning; its symbolic execution also has strong similarities with IVL back-ends.

Maksimović et al. [68] briefly describe a soundness framework for GIL [63], a parametric program representation used by the Gillian project. GIL needs to be instantiated with a state model, primitive assertions, and memory actions to obtain specific intermediate representations (essentially, multiple IVLs) useful for different verification projects (e.g., for JavaScript [63] and Rust [64]). However, each GIL instantiation also determines the back-end verification algorithm. As such, there is no common semantics that abstracts over different verification algorithms.

[68]: Maksimović et al. (2021), *Gillian*

[63]: Maksimović et al. (2021), *Gillian, Part II*

[63]: Maksimović et al. (2021), *Gillian, Part II*

[64]: Ayoun et al. (2025), *A Hybrid Approach to Semi-automated Rust Verification*

This work. In this chapter, we present a framework for formally justifying translational separation logic verifiers. At its center is a generic IVL, called CoreIVL, that captures the essence of different IVLs based on separation logics. In particular, CoreIVL can be instantiated with different statements, assertion languages, and separation algebras; we introduce in particular the novel concept of an *IDF algebra*, which generalizes the notion of separation algebra to also model the implicit dynamic frames logic used in Viper.

To address Challenge 1 above, we define the semantics of CoreIVL (and correspondingly, each of its instantiations) using *dual (*i.e.*, demonic and angelic) non-determinism*. Demonic non-determinism is a standard technique to verify properties for all inputs, thread schedules, etc. Our novel insight is to complement it with angelic non-determinism to abstract over the different proof search algorithms employed by back-ends. Intuitively, the IVL program is correct if *any* of these algorithms succeeds, which is an angelic behavior.

To address Challenge 2, we define an operational semantics for CoreIVL, which incorporates these notions of dual non-determinism and, like CoreIVL itself, is parametric in an IDF algebra, to support both separation logic and IDF. An *operational* semantics facilitates proving a formal connection to the concrete verification algorithms used in back-ends. Separation logic verifiers typically perform symbolic execution, which is typically described operationally [150] and (as we show) can be connected to our operational semantics via a standard simulation proof. Similarly, an operational IVL semantics is well-suited for formalizing the connections to back-ends that encode IVL programs into a further, more basic IVL, such as Viper's verification condition generator, which encodes Viper programs into Boogie.

[150]: de Boer et al. (2021), *Symbolic Execution Formally Explained*

To address Challenge 3, we define an axiomatic semantics for CoreIVL and prove its equivalence to our operational semantics. An *axiomatic* semantics facilitates proving a formal connection to the program logic used on the front-end level because both deal with derivations, which are often structurally related due to the compositional nature of most IVL translations. In addition, we are able to prove some powerful generic results about idiomatic encoding patterns once-and-for-all, further minimizing the instantiation-specific gap that a formal soundness proof needs to bridge.

We illustrate the practical applicability of our formal framework by instantiating CoreIVL with elements of Viper. We use the resulting operational semantics to prove the soundness of two verification back-ends: a formalization of the central features of Viper's symbolic execution back-end, and a pre-existing formalization of Viper's verification condition generator not presented in this thesis [74]. These proofs demonstrate, in particular, that our use of angelic non-determinism allows us to capture these two rather disparate (and representative) back-ends. At the other end, we prove soundness of a front-end based on concurrent separation logic using our axiomatic semantics. These proofs demonstrate that our framework effectively closes the large semantic gap between front-ends and back-ends and enables formal reasoning about the entire chain.

[74]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers*

Contributions and outline. We make the following technical contributions:

- We introduce a novel notion of *IDF algebra*, which generalizes the notion of separation algebra to capture both separation logic and implicit dynamic frames.
- We present a formal framework for reasoning about translational separation logic verifiers, via a parametric language CoreIVL, for which we define a novel operational semantics combining core separation-logic reasoning principles and dual non-determinism. We define an alternative axiomatic semantics, and show its equivalence with our operational semantics.
- We define a Viper instantiation of CoreIVL. We formalize and prove the soundness of the core of Viper's symbolic execution back-end. Similarly, we show soundness of an existing formalization of Viper's back-end based on verification condition generation. These proofs illustrate how angelic non-determinism can abstract over these different algorithmic choices.

- We formalize a front-end for a simple concurrent language to be verified with concurrent separation logic, as well as its standard encoding as employed in translational verifiers, and prove this encoding sound with respect to our axiomatic semantics for CoreIVL.

We give an overview of our key ideas in Section 2.2. We define the operational and axiomatic IVL semantics in Section 2.3. We discuss how to prove back-end soundness in Section 2.4 and front-end soundness in Section 2.5, and we discuss related work in Section 2.6.

All formalizations and proofs in this chapter are mechanized in the Isabelle proof assistant [33] and our mechanization is publicly available [129].

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

[129]: Dardinier et al. (2024), *Formal Foundations for Translational Separation Logic Verifiers – Artifact*

2.2. Key Ideas

In this section, we present the key ideas behind our work. Our framework and its instantiation to Viper is presented in Figure 2.1. At its center is *CoreIVL* (depicted by the grey area), a general core language for representing SL-based IVLs. This core language bridges the substantial gap between proofs of high-level programs using custom verification logics (*e.g.*, *concurrent separation logic* [11] (CSL) in the figure) at the front-end level and verification algorithms for SL-based IVLs at the back-end level (*e.g.*, symbolic execution and verification condition generation). CoreIVL is *parametric* in its state model and assertions, so that it can represent multiple variants of separation logic (*e.g.*, those on which VeriFast and GIL are based), including implicit dynamic frames (on which Viper is based). In Figure 2.1, ViperCore represents the instantiation of these parameters for Viper. We give two equivalent semantics to CoreIVL: An *operational* semantics, which is designed to enable soundness proofs for diverse back-end verification algorithms (shown on the right of Figure 2.1), and an *axiomatic* semantics, which can be used to prove front-end translations into CoreIVL sound, by connecting this axiomatic semantics to the front-end separation logic (shown on the left).

[11]: O'Hearn (2007), *Resources, Concurrency, and Local Reasoning*

The rest of this section is organized as follows. Section 2.2.1 introduces the general core language *CoreIVL* for representing SL-based IVLs. Section 2.2.2 illustrates how to check for the existence of a Concurrent Separation Logic front-end proof for a parallel program by encoding the verification problem into our sequential CoreIVL, mimicking the approach of modern translational verifiers. Section 2.2.3 presents the formal operational semantics of CoreIVL. Finally, Section 2.2.4 presents an alternative equivalent *axiomatic semantics* for CoreIVL, and shows how it can be leveraged to prove a front-end translation sound.

2.2.1. A Core Language for SL-Based IVLs

In this section, we first motivate and then define a core language for SL-based IVLs, called *CoreIVL*, which captures central aspects of SL-based verifiers, such as Viper [16], Gillian [30, 63], or VeriFast [15].

[16]: Müller et al. (2016), *Viper*

[30]: Fragoso Santos et al. (2020), *Gillian, Part i*

[63]: Maksimović et al. (2021), *Gillian, Part II*

[15]: Jacobs et al. (2011), *VeriFast*

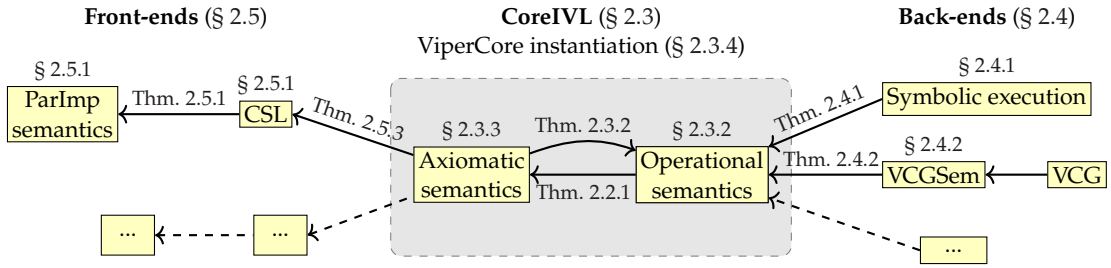

Figure 2.1.: Overview of our framework and its application to Viper. The yellow boxes represent components of our framework (such as semantics and logics), while the arrows show the theorems that connect them. The dashed arrows and the unlabeled yellow boxes represent potential additional front-ends and back-ends that could be connected to ViperCore. CSL stands for concurrent separation logic, and VCG for verification condition generation. We have formally connected VCG to VCGSem in work not presented in this thesis [74].

Manipulating SL states via inhale and exhale. At the core of these verifiers is the SL state they track throughout the verification, typically containing a heap (a mapping from heap locations to values) and SL resources (such as fractional permissions to heap locations). This SL state is manipulated with two verification primitives: `inhale` A (also called *assume** and *produce*) and `exhale` A (also called *assert** and *consume*), where A is a separation logic assertion. `inhale` A assumes the logical constraints in A (e.g., constraints on integer values), and adds the resources (e.g., ownership of heap locations) specified by A to the current state. Dually, `exhale` A asserts that the logical constraints in A hold, and removes the resources specified by A from the current state. These two primitives can encode the verification conditions for a wide variety of program constructs. For instance, a procedure call is encoded as exhaling the call's precondition (to check its logical constraints and transfer ownership of resources from caller to callee), followed by inhaling the postcondition (to assume logical constraints and gain resources back from the call).

Diversity of logics and their semantics. While SL-based IVLs all employ some version of these two inhale and exhale primitives, their actual logics are surprisingly diverse in both core connectives and their semantics. GIL and VeriFast support different separation logics, while Viper uses implicit dynamic frames (IDF), a variation of separation logic that allows for heap-dependent expressions in assertions (e.g., separation logic's points-to predicate $e.f \mapsto v$ is expressed as $\mathbf{acc}(e.f) * e.f = v$ in IDF, in which the ownership of the heap location and a logical constraint on its value are expressed as two separate conjuncts)[2].

IVLs also support different SL connectives: Viper supports iterated separating conjunctions [151], Viper and VeriFast support fractional recursively-defined predicates [81, 140] (which will be discussed in Chapter 3), Viper and Gillian support magic wands [79, 141] (which will be discussed in Chapter 4[3]), and VeriFast supports arbitrary existential quantification.

A standard approach for generic reasoning over large classes of separation logics is to build reasoning principles based on a *separation algebra* (built over a partial commutative monoid) [85, 86]. We extend this classic concept to a novel notion of *IDF algebra*, which can model separation logics and IDF alike. In particular, IDF algebras allow asserting knowledge about the value of heap locations $e.f$ without asserting ownership of the heap location itself.

2: This difference also affects the semantic models; separation logic is typically formalized using partial heaps, whereas IDF typically uses a total heaps model [87].

[151]: Müller et al. (2016), *Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution*

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

[140]: Dardinier et al. (2022), *Fractional Resources in Unbounded Separation Logic*

[79]: Schwerhoff et al. (2015), *Lightweight Support for Magic Wands in an Automatic Verifier*

[141]: Dardinier et al. (2022), *Sound Automation of Magic Wands*

3: The support for magic wands in Gillian is based on Chapter 4 of this thesis.

[85]: Calcagno et al. (2007), *Local Action and Abstract Separation Logic*

[86]: Dockins et al. (2009), *A Fresh Look at Separation Algebras and Share Accounting*

$$C ::= \texttt{inhale}\ A \mid \texttt{exhale}\ A \mid \texttt{havoc}\ x \mid C;C \mid \texttt{if}\ (b)\ \{C\}\ \texttt{else}\ \{C\} \mid x := e \mid \texttt{skip} \mid \texttt{custom}\ C'$$

Figure 2.2.: Syntax of statements in CoreIVL. A is an assertion, x a variable, b a Boolean expression, e an arbitrary expression. Assertions and expressions are represented semantically as sets of states and partial functions from states to values, respectively. C' represents custom statements and is a parameter of the language.

```
method main(p: Cell)
  // requires acc(p.v, _)
{
  q := alloc(0)


  // {P_l}          {P_r}
  q.v := p.v || tmp := p.v
  // {Q_l}          {Q_r}

  tmp := tmp + q.v

  free(q)

  assert tmp = p.v + p.v
}
```

```
method main_ivl(p: Ref) {
  inhale acc(p.v, _)

  havoc q
  inhale acc(q.v) * q.v = 0

  exhale P_l * P_r
  havoc tmp
  inhale Q_l * Q_r

  tmp := tmp + q.v

  exhale acc(q.v)

  exhale tmp = p.v + p.v
}
```

```
method l(p,q:Ref){
  inhale P_l
  q.v := p.v
  exhale Q_l
}

method r(p,q:Ref){
  inhale P_r
  tmp := p.v
  exhale Q_r
}
```

Figure 2.3.: A simple parallel program (left), annotated with a method precondition, as well as pre- and postconditions for the parallel branches, and its encoding into CoreIVL (instantiated to model Viper), consisting of a main IVL method (middle) and two further methods (right) modeling the parallel branches (that is, the premises of CSL's parallel composition rule). We use the shorthands $P_l \triangleq \texttt{acc}(p.v, \_) * \texttt{acc}(q.v)$, $Q_l \triangleq \texttt{acc}(p.v, \_) * \texttt{acc}(q.v) * \texttt{p.v} = \texttt{q.v}$, $P_r \triangleq \texttt{acc}(p.v, \_)$, and $Q_r \triangleq \texttt{acc}(p.v, \_) * \texttt{tmp} = \texttt{p.v}$, where the IDF assertion $\texttt{acc}(e, \_)$ expresses non-zero permission to e (corresponding to the SL assertion $\exists p, v.\, e \overset{p}{\mapsto} v$).

Core Language. The syntax of CoreIVL is shown in Figure 2.2. To capture the diversity of assertions supported in existing SL-based IVLs, assertions A in our core language are *semantic*, *i.e.*, assertions are *sets of states* (as opposed to fixing a syntax, and having the semantics for this syntax determine the set of states in which a syntactic assertion is true);[4] states themselves are taken from any chosen IDF algebra. Similarly, expressions e are semantically represented as partial functions from states to values. Moreover, although we assume some core statements in our language, we allow these to be arbitrarily extended via a parameter for *custom statements* C', for instance, to add field assignments. The statements of our core language contain the key verification primitives `inhale` and `exhale` described above, as well as `havoc`, which non-deterministically assigns a value to a variable. Combined with conditional branching, `inhale`, `exhale`, and `havoc` allow us to encode many important statements, such as while loops, procedure calls, and even proof rules for parallel programs, as we show in the next subsection.

4: In Chapter 3, we will present a syntax for separation logic assertions, and show how to interpret them.

2.2.2. Background: Translational Verification of a Parallel Program

We use the parallel program on the left in Figure 2.3 to illustrate how translational verification works, and the challenges that arise in formalizing this widely-used approach. This program takes as input a Cell `p` (an object with a value field `v`), allocates a new Cell `q`, assigns the value of `p.v` in parallel to `q.v` and to the variable `tmp`, then adds the value of

`q.v` to `tmp`, deallocates `q`, and finally asserts that `tmp` is equal to `p.v + p.v`. Our goal is to verify this program in Concurrent Separation Logic (CSL) [11], that is, by encoding the program and the proof rules of CSL into CoreIVL. In particular, we want to prove that the assertion on its last line holds.

[11]: O'Hearn (2007), *Resources, Concurrency, and Local Reasoning*

Although the original CSL is presented via standard separation logic syntax, we use the syntax of IDF to annotate this example. The syntax **acc**(e.v, f) denotes *fractional permission* (ownership) of the heap location `e.v` (where $f = 1$ allows reading and writing, and a fraction $0 < f < 1$ allows reading) [81]. The syntax **acc**(p.v, _) (used as precondition in our example) denotes a so-called *wildcard permission* (or *wildcard* in short); it is shorthand for $\exists f > 0.$ **acc**(p.v, f), which guarantees read access while abstracting the precise fraction.

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

Correctness of our example means proving a CSL triple $\Delta \vdash_{CSL} [\textbf{acc}(\texttt{p.v}, \_)]\ C\ [\top]$, where C is the body of the method `main` in the front-end (left) program ($\top$ is the trivial postcondition). Instead of constructing a proof directly, a translational verifier maps this to an IVL program (shown as a CoreIVL program to the middle and right of Figure 2.3) whose correctness implies the existence of a CSL proof for the original program.

Encoding the program into CoreIVL. Our encoding models each proof task of the CSL verification problem as a separate IVL method, whose statements reflect the individual proof steps [152]. The IVL methods `main_ivl`, `l` and `r` are constructed such that the correctness of *all three* implies the existence of a valid CSL proof for `main`.

[152]: Leino et al. (2009), *A Basis for Verifying Multi-threaded Programs*

The precondition **acc**(p.v, _) of `main` is modeled by the first **inhale** statement in `main_ivl`, reflecting that the proof of the main method may rely on the resources and assumptions guaranteed by this precondition. The allocation `q := alloc(0)` is then encoded via a **havoc** and an **inhale** statement to non-deterministically choose a memory location and obtain a full (*i.e.*, 1) permission. Dually, the deallocation **free**(q) after the parallel composition is encoded via an **exhale** statement, which removes this (full) permission from the IVL state. Since permissions are non-duplicable (technically, affine) resources, this encoding guarantees that no permission can remain and so any attempt to later access this location would cause a verification failure.

To understand the encoding of a source-level parallel composition, we recall the CSL proof rule[5]:

$$\textsc{Par}\ \frac{\Delta \vdash_{CSL} [P_l]\ C_l\ [Q_l] \qquad \Delta \vdash_{CSL} [P_r]\ C_r\ [Q_r]}{\Delta \vdash_{CSL} [P_l * P_r]\ C_l \mid\mid C_r\ [Q_l * Q_r]}$$

5: We omit technical side-conditions from the original rule that restrict mutation of variables shared amongst threads; these are taken care of properly in real verifiers and our formalizations.

From the point of view of the *outer thread* (forking and joining the parallel branches), the overall effect of the parallel composition can be seen as *giving up* the separating conjunction $P_l * P_r$ of the preconditions of the parallel branches, and obtaining the corresponding postconditions $Q_l * Q_r$ before resuming any remaining code[6]. This exchange of assertions across the triple in the conclusion of the rule (as well as the intervening modification of `tmp`) is modeled in the IVL program by the sequence **exhale** $P_l * P_r$; **havoc** `tmp`; **inhale** $Q_l * Q_r$.

6: We assume (as is common for modular verifiers) that each thread's specification is explicitly annotated, as in `main`.

$$\textsc{InhaleOp}\quad \langle \texttt{inhale}\ A, \omega\rangle \rightarrow_\Delta \{\omega\} * A$$

$$\textsc{ExhaleOp}\quad \frac{\omega = \omega' \oplus \omega_A \qquad \omega_A \in A}{\langle \texttt{exhale}\ A, \omega\rangle \rightarrow_\Delta \{\omega'\}}$$

$$\textsc{SeqOp}\quad \frac{\langle C_1, \omega\rangle \rightarrow_\Delta S_1 \qquad \forall \omega_1 \in S_1.\, \langle C_2, \omega_1\rangle \rightarrow_\Delta \mathcal{S}(\omega_1)}{\langle C_1; C_2, \omega\rangle \rightarrow_\Delta \cup_{\omega_1 \in S_1} \mathcal{S}(\omega_1)}$$

(a) Selected operational semantics rules.

$$\textsc{InhaleAx}\quad \Delta \vdash [P]\ \texttt{inhale}\ A\ [P * A]$$

$$\textsc{ExhaleAx}\quad \frac{P \models Q * A}{\Delta \vdash [P]\ \texttt{exhale}\ A\ [Q]}$$

$$\textsc{SeqAx}\quad \frac{\Delta \vdash [P]\ C_1\ [R] \qquad \Delta \vdash [R]\ C_2\ [Q]}{\Delta \vdash [P]\ C_1; C_2\ [Q]}$$

(b) Selected axiomatic semantic rules.

Figure 2.4.: Selected simplified operational and axiomatic semantic rules.

The premises of the parallel rule are checked by verifying two extra methods `l` and `r`, whose pre- and postconditions correspond to the Hoare triples from the rule premises directly. The encoded bodies of `l` and `r` follow the standard pattern: an **`inhale`** of their preconditions (which can be seen as the other "half" of the transfer from the outer thread, modeled by **`exhale`** $P_l * P_r$), the translation of their source implementations, and finally an **`exhale`** of their postconditions.

If running a back-end verifier for the IVL on the three encoded methods succeeds, we have demonstrated that a CSL proof for the original program exists—provided that the translational verification is sound. Soundness depends on a non-trivial translation, the subtle semantics of an IVL, and the algorithms employed by back-end verifiers. In the rest of this section, we explain our formal framework for establishing the soundness of translational verifiers.

2.2.3. Operational Semantics and Back-End Verifiers

To make formal claims about an IVL program, we need a formal semantics and notion of correctness for the IVL itself. As explained in the introduction, an operational semantics facilitates a formal connection to various back-end algorithms, which typically have an operational flavor. Since our semantics needs to capture verification algorithms that make heavy use of (demonic) non-determinism (to model concurrency, allocation, or abstract modularly over the precise behavior of program elements), our operational semantics embraces such non-determinism. Moreover, to account for the diversity of the verification algorithms used in back-ends, our semantics also incorporates the dual notion of *angelic non-determinism*.

Consider verifying the statement **`exhale`** **`acc`**`(a.v)` $\vee$ **`acc`**`(b.v)`, which requires giving up (full) permission to *either* `a.v` or `b.v`; if the original state holds both permissions, either choice avoids a failure here, but results in different successor states, and so might affect whether subsequent statements verify successfully. Such algorithmic choices occur for other IVL constructs, such as for choosing the values of existentials (including the amount of permission for a wildcard permission), or determining the footprints of magic wands (as we will see in Chapter 4). Our operational semantics makes *all algorithmic choices possible* and defines a program as correct if *any* such choice avoids failure.

Operational semantics. To capture the dual non-determinism, we define our operational semantics as a multi-relation [153, 154]

$$\langle C, \omega \rangle \rightarrow_{\Delta} S$$

where C is an IVL statement, ω an initial state, S a set of final states, and Δ a type context (mapping for example variables to types, *i.e.*, to sets of values). The set S captures the *demonic choices*, *i.e.*, contains the resulting state for each possible demonic choice. On the other hand, *angelic choices* are reflected by *different result sets* derivable in our semantics. Returning to our previous example, if ω is a state with full permission to both `a.v` and `b.v`, our semantics allows for both transitions $\langle$**exhale** **acc**(`a.v`) $\vee$ **acc**(`b.v`), $\omega\rangle \rightarrow_{\Delta} \{\omega_{-a}\}$ and $\langle$**exhale** **acc**(`a.v`) $\vee$ **acc**(`b.v`), $\omega\rangle \rightarrow_{\Delta} \{\omega_{-b}\}$ (where ω_{-a} and ω_{-b} are identical to ω but with the permission to `a.v` resp. `b.v` removed).

A successful verification by a back-end is represented by an execution in our operational semantics, leading to the following definition of correctness of a CoreIVL statement:

Definition 2.2.1 Correctness and validity of CoreIVL statements**.**
*A CoreIVL statement C is **correct** for an initial state ω iff C executes successfully in ω,* i.e., $\exists S. \langle C, \omega\rangle \rightarrow_{\Delta} S$.
*C is **valid** iff it is correct for all well-typed[7] stable initial states.*

Figure 2.4a shows simplified rules for the operational semantics of **inhale** A, **exhale** A, and sequential composition. The (non-simplified) rules for all statements are shown in Section 2.3. Inhaling A in state ω leads to the set of all possible combinations $\omega \oplus \omega_A$ for $\omega_A \in A$, capturing the demonic non-determinism of **inhale**: All possible states satisfying A must be considered in the rest of the program. Dually, the rule ExhaleOp allows *any choice of state* ω_A satisfying A (that is, uses angelic non-determinism), and to remove it from ω. In our previous example, ω can be decomposed into $\omega = \omega_{-a} \oplus \omega_a$ or $\omega = \omega_{-b} \oplus \omega_b$, where ω_a and ω_b respectively contain the permission to `a.v` and `b.v` (and thus ω_a and ω_b both satisfy the exhaled assertion **acc**(`a.v`) $\vee$ **acc**(`b.v`)). The rule SeqOp for sequential composition is more involved, since it needs to deal with the dual non-determinism: It requires a single function $\mathcal{S}$ that maps every state ω_1 from S_1 (the set of states obtained after executing C_1 in ω) to a set of states $\mathcal{S}(\omega_1)$ that can be reached by executing C_2 in ω_1. The choice of the function $\mathcal{S}$ captures the angelic choices in C_2.

Connection to back-end verifiers. To show that this operational semantics for CoreIVL is indeed suitable to capture different verification algorithms, we connect it to formalizations of the two main back-ends used by Viper. First, we formalize a version of Viper's symbolic execution back-end [155] in Isabelle/HOL and prove it sound against the operational semantics of CoreIVL. Second, we connect the formalization of Viper's verification condition generation back-end by Parthasarathy et al. [74] to CoreIVL by constructing a CoreIVL execution from a successful verification by their back-end. The soundness proofs of these back-ends are described in Section 2.4. There we will also see that the angelic choice described earlier in this section is crucial for enabling these proofs since

[153]: Rewitzky (2003), *Binary Multirelations*
[154]: Guéneau et al. (2023), *Melocoton*

7: We do not discuss typing in details in this chapter, but our Isabelle formalization covers it.

[155]: Schwerhoff (2016), *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*

[74]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers*

the two back-ends use different heuristics, in particular around exhaling wildcard permissions.

2.2.4. Axiomatic Semantics

The previously-introduced definition of correctness (Definition 2.2.1) based on the operational semantics is well-suited to connect to back-end verifiers. However, connecting it to front-end programs, and especially logics such as CSL in our example from Figure 2.3, requires substantial effort due to the large semantic gap between the operational IVL semantics and the front-end logic. The IVL semantics presented previously is *operational*, describes the execution from a *single state*, and exposes low-level details (such as handling the dual non-determinism in the rule SeqOp). In contrast, the program logic is *axiomatic*, describes the behavior of *sets of states* (via assertions), and is more high-level (*e.g.*, it uses an intermediate assertion in the rule SeqAx instead of the semantic function $\mathcal{S}$). To bridge this gap, we present an alternative (and, as we later prove, equivalent) axiomatic semantics for CoreIVL, which is closer to the separation logics typically used for front-end programs and, thus, simplify the proof that a front-end translation is sound.

Our axiomatic semantics uses judgments of the form

$$\Delta \vdash [P]\ C\ [Q]$$

where P and Q are semantic assertions (sets of states), C is an IVL statement, and Δ is a type context. Intuitively, this triple expresses that C can be executed successfully in any state from P (with the right angelic choices), and Q is (precisely) the set of all states reached by these executions. Formally, we want the following *soundness* property (we will present the completeness theorem in Section 2.3):

Theorem 2.2.1 Operational-to-axiomatic soundness**.**
If the CoreIVL statement C is well-typed and valid (Definition 2.2.1) then there exists a set of states B such that $\Delta \vdash [\top]\ C\ [B]$ holds.

Note that, in contrast to when one defines a proof system for a pre-existing operational semantics, the desired implication here is from operational to axiomatic semantics; this is due to the connection we are aiming for from back-end algorithms (defined operationally) to front-end proofs.

The rules for the axiomatic semantics of **`inhale`** A, **`exhale`** A, and sequential composition are shown in Figure 2.4b. The rule InhaleAx for **`inhale`** A corresponds to the operational rule InhaleOp, where ω has been lifted to the set of states P (since $P * A = \bigcup_{\omega \in P}(\{\omega\} * A)$). The rule ExhaleAx for **`exhale`** A is more involved, as it first requires weakening the set of initial states P to $Q * A$. Weakening is in general necessary to disentangle the states in Q and A: For example, to exhale **`acc`**`(a.v)` from a precondition **`acc`**`(a.v)` $*$ **`acc`**`(b.v)` $*$ `a.v = b.v`, we have to first drop the equality `a.v = b.v` because otherwise the resulting postcondition would refer to a memory location that is no longer owned. Moreover, similarly to how Hoare logic hides the induction necessary to reason about unbounded while loops behind a loop invariant, our axiomatic semantics hides the dual non-determinism of the operational semantics behind

$$\textsc{Frame}\ \frac{\Delta \vdash_{CSL} [P]\ C\ [Q] \qquad fv(F) \cap mod(C) = \emptyset}{\Delta \vdash_{CSL} [P * F]\ C\ [Q * F]} \qquad \textsc{Par}\ \frac{\Delta \vdash_{CSL} [P_l]\ C_l\ [Q_l] \qquad \Delta \vdash_{CSL} [P_r]\ C_r\ [Q_r] \qquad \ldots}{\Delta \vdash_{CSL} [P_l * P_r]\ C_l \mid\mid C_r\ [Q_l * Q_r]}$$

$$\textsc{Seq}\ \frac{\Delta \vdash_{CSL} [P]\ C_1\ [R] \qquad \Delta \vdash_{CSL} [R]\ C_2\ [Q]}{\Delta \vdash_{CSL} [P]\ C_1; C_2\ [Q]} \qquad \textsc{Cons}\ \frac{\Delta \vdash_{CSL} [P']\ C\ [Q'] \qquad P \models P' \qquad Q' \models Q}{\Delta \vdash_{CSL} [P]\ C\ [Q]}$$

$$\textsc{Alloc}\ \frac{r \notin fv(e)}{\Delta \vdash_{CSL} [\top]\ \mathtt{r} := \mathtt{alloc}(e)\ [\mathtt{acc}(\mathtt{r.v}) * r.v = e]} \qquad \textsc{Free}\ \Delta \vdash_{CSL} [\mathtt{acc}(\mathtt{q.v})]\ \mathtt{free(q)}\ [\top]$$

Figure 2.5.: Selected CSL rules. In the rule Frame, $fv(F)$ and $mod(C)$ denote the set of variables free in F and the set of variables potentially modified by C, respectively.

high-level connectives such as the separating conjunction. Intuitively, in the rule ExhaleAx, the angelic choice is hidden in the choice of Q and in the split of every state in P into a state in Q and a state in A. In our previous example `exhale acc(a.v) ∨ acc(b.v)`, we could choose Q to be either `acc(a.v)` or `acc(b.v)`, *i.e.*, we could derive both $\Delta \vdash [\mathtt{acc(a.v)} * \mathtt{acc(b.v)}]\ \mathtt{exhale\ acc(a.v) \vee acc(b.v)}\ [\mathtt{acc(a.v)}]$ and $\Delta \vdash [\mathtt{acc(a.v)} * \mathtt{acc(b.v)}]\ \mathtt{exhale\ acc(a.v) \vee acc(b.v)}\ [\mathtt{acc(b.v)}]$.

Finally, the rule SeqAx for sequential composition illustrates how the axiomatic semantics abstracts over the low-level details of the dual non-determinism in the operational semantics, such as the existence of the semantic function $\mathcal{S}$ in rule SeqOp. Instead, the axiomatic rule SeqAx uses an intermediate assertion R; its relation to $\mathcal{S}$ is proved once and for all in the soundness proof and, thus, does not have to be proved for each front-end.

Crucially, we have designed the axiomatic semantics such that it contains *exactly one rule* per statement. In particular, it contains no structural rules such as a frame rule or a consequence rule, which are not necessary in our setting. This allows us to deconstruct an axiomatic semantic derivation into smaller blocks, to then reconstruct a proof in the front-end logic. For example, one can derive from $\Delta \vdash [P]\ C_1; C_2\ [Q]$ the existence of some assertion R such that $\Delta \vdash [P]\ C_1\ [R]$ and $\Delta \vdash [R]\ C_2\ [Q]$ hold. Using this axiomatic semantics, we can now easily connect the correctness of the IVL program to the correctness of the front-end program, as we explain next.

Connecting to front-end programs and logics. Let us now see how the axiomatic semantics enables us to construct a CSL proof for the front-end program from Figure 2.3. Concretely, we build a CSL proof of the triple $\Delta \vdash_{CSL} [\mathtt{acc(p.v, \_)}]\ C\ [\top]$, where C corresponds to the body of the method `main`. To do this, we use the CSL rules shown in Figure 2.5 and the CoreIVL triples $\Delta \vdash [\top]\ C\ [B]$ for the methods `l`, `r`, and `main_ivl` that we obtain from Theorem 2.2.1.

The first step of proving the CSL triple for `main` is to pair each statement in `main` with the corresponding code in `main_ivl`. For this, we use CSL's Seq rule and (the inversion of) SeqAx to split the proofs for `main` and

`main_ivl` into smaller parts:

	$\Delta \vdash [\top]$ **inhale acc**(p.v, _) $[A_0]$
$\Delta \vdash_{CSL} [A_0]$ `q := alloc(0)` $[A_1]$	$\Delta \vdash [A_0]$ **havoc** q;**inhale acc**(q.v) * q.v = 0 $[A_1]$
$\Delta \vdash_{CSL} [A_1]$ `q.v := p.v || tmp := p.v` $[A_2]$	$\Delta \vdash [A_1]$ **exhale** P_l*P_r; **havoc** tmp; **inhale** Q_l*Q_r $[A_2]$
$\Delta \vdash_{CSL} [A_2]$ `tmp := tmp + q.v` $[A_3]$	$\Delta \vdash [A_2]$ `tmp := tmp + q.v` $[A_3]$
$\Delta \vdash_{CSL} [A_3]$ `free(q)` $[A_4]$	$\Delta \vdash [A_3]$ **exhale acc**(q.v) $[A_4]$
$\Delta \vdash_{CSL} [A_4]$ **assert** `tmp = p.v + p.v` $[B]$	$\Delta \vdash [A_4]$ **exhale** `tmp = p.v + p.v` $[B]$

Note how deconstructing the applications of SeqAx in the proof of `main_ivl` gives us intermediate assertions A_{0-4}, which we use to instantiate the intermediate assertion R in Seq.[8] Matching statements of the front-end program to segments of the CoreIVL program is straightforward since the front-end translation is typically defined statement by statement.

After decomposing the sequential compositions, we justify the CSL triple for each primitive front-end statement from the corresponding CoreIVL triple. For some statements like `tmp := tmp + q.v`, this is trivial as the triples (and corresponding logic rules) match. Let us now focus on the most interesting cases: `q := alloc(0)`, `q.v := p.v || tmp := p.v`, and `free(q)`.

8: Note that the CSL we use in this chapter has the same state model as the IVL, and thus the IVL assertions do not need to be converted to CSL assertions. Our axiomatic semantics can also be used to reconstruct proofs in program logics with different state models, but this goes beyond the scope of this thesis.

The exhale-havoc-inhale pattern. To derive the CSL triples for these statements, we observe that their encoding follows a pattern: The CoreIVL code first exhales the precondition P of the CSL rule (omitted if $P = \top$), then havocs the variables modified by the statement (`q` for `q := alloc(0)` and `tmp` for `q.v := p.v || tmp := p.v`), and finally inhales the postcondition Q of the CSL rules (omitted if $Q = \top$), leading to the pattern **exhale** P;**havoc** x_1;...;**havoc** x_n;**inhale** Q. To handle this general pattern, we can use the following lemma, which holds for any separation logic $\mathcal{L}$ with a consequence rule and a frame rule (see Section 2.5 for the proof):

Lemma 2.2.2 Exhale-havoc-inhale pattern.
For any separation logic $\mathcal{L}$ that has a frame rule and a consequence rule, if $\Delta \vdash [A]$ **exhale** P;**havoc** x_1;...;**havoc** x_n;**inhale** Q $[B]$ *holds and* $\Delta \vdash_{\mathcal{L}} [P]\ C\ [Q]$ *holds, where* $\{x_1, \ldots, x_n\} = mod(C)$, *then* $\Delta \vdash_{\mathcal{L}} [A]\ C\ [B]$ *holds.*

Intuitively, this lemma shows that a CoreIVL triple for the exhale-havoc-inhale pattern allows us to obtain the corresponding CSL triple. In the case of `q := alloc(0)`, this lets us lift Alloc to the precondition A_0 and postcondition A_1, giving us exactly the triple we need. To justify the triple for `q.v := p.v || tmp := p.v`, we need to establish the premises of the rule Par, $\Delta \vdash_{CSL} [P_l]$ `q.v := p.v` $[Q_l]$ and $\Delta \vdash_{CSL} [P_r]$ `tmp := p.v` $[Q_r]$, which can be derived from the correctness of the methods `l` and `r` using a lemma similar to Lemma 2.2.2, as we formally show in Section 2.5.

Summary. We have now seen how to justify the translational verification of the program from Figure 2.3 in CSL in three steps. First, we showed that the successful verification of its CoreIVL encoding in a back-end implies that the CoreIVL program is valid. Second, we used the soundness

theorem for the axiomatic IVL semantics to derive judgments in the axiomatic semantics. Third, we use those judgments to prove the desired CSL triple. Each of these steps is well-suited for its task: The operational semantics allows us to connect to the back-end verifiers, while the axiomatic semantics facilitates the reconstruction of the front-end logic proof—both linked by Theorem 2.2.1.

2.3. Semantics

In this section, we present an operational and an axiomatic semantics for the CoreIVL language defined in Figure 2.2. We first define in Section 2.3.1 an IDF algebra that captures both separation logic and implicit dynamic frames state models. We then formalize the operational semantics of CoreIVL in Section 2.3.2 and define its axiomatic semantics and prove their equivalence in Section 2.3.3. We instantiate CoreIVL for key features of Viper in Section 2.3.4.

2.3.1. An Algebra for Separation Logic and Implicit Dynamic Frames

A standard way to capture different separation logic state models is to use a *separation algebra* [85, 86], *i.e.*, a partial commutative monoid $(\Sigma, \oplus)$, where Σ is the set of all states, and $\oplus$ is a partial, commutative, and associative binary operator, used to combine states (*e.g.*, via the separating conjunction operator $*$). In SL, assertions about values of heap locations must also assert ownership of those heap locations. In particular, asserting that a heap location `x.f` has the value 5 requires using the points-to predicate $\texttt{x.f} \mapsto 5$, which also expresses ownership of the location `x.f`. This requirement is embedded in the SL state model. For example, a typical SL state with a heap and fractional permissions (ignoring local variables for now) is $\Sigma_{SL} \triangleq (L \rightharpoonup (V \times (0, 1]))$, *i.e.*, a partial function from a set L of heap locations to pairs of values from a set V and positive fractional permissions. That is, any value for a heap location is associated with a strictly positive permission.

[85]: Calcagno et al. (2007), *Local Action and Abstract Separation Logic*
[86]: Dockins et al. (2009), *A Fresh Look at Separation Algebras and Share Accounting*

In contrast, in implicit dynamic frames, an assertion may constrain the value of a heap location independently of expressing ownership. For example, $\texttt{x.f} = 5$ is a valid IDF assertion that expresses that `x.f` stores the value 5 without expressing ownership of `x.f`. However, IDF requires assertions used as pre- and postconditions, loop invariants, frames (for the frame rule), etc. to be *self-framing*, that is, to express ownership of all heap locations they mention. For example, $\texttt{acc}(\texttt{x.f}) * \texttt{x.f} = 5$ is self-framing, while $\texttt{x.f} = 5$ is not. To capture IDF states with fractional permissions, we define the state model $\Sigma_{IDF} \triangleq (L \rightharpoonup V) \times (L \rightharpoonup [0, 1])$.[9] In contrast to Σ_{SL}, values and permissions are separated in Σ_{IDF}, which allows states (h, π) where $h(\texttt{x.f}) = 5$ but $\pi(\texttt{x.f}) = 0$.

We call a state $(h, \pi) \in \Sigma_{IDF}$ *stable* iff it contains values exactly for the heap locations with non-zero permission, *i.e.*, $dom(h) = \{l \mid \pi(l) > 0\}$. Stable states are exactly those that can be represented as states in Σ_{SL}; By construction, all states in Σ_{SL} are stable.

9: In addition, values must exist for those heap locations where the state has non-zero permission. That is, Σ_{IDF} is restricted to states (h, π) such that $\forall l.\, \pi(l) > 0 \Rightarrow l \in dom(h)$. Alternatively, one could avoid this restriction by defining the state model as $\Sigma_{IDF} \triangleq (L \rightharpoonup (V \times [0, 1]))$; the only difference with Σ_{SL} is that we use the interval $[0, 1]$ instead of $(0, 1]$ for permission amounts.

$$a \oplus b = b \oplus a \qquad a \oplus (b \oplus c) = (a \oplus b) \oplus c \qquad c = a \oplus b \land c = c \oplus c \Rightarrow a = a \oplus a$$

$$x = x \oplus |x| \qquad |x| = |x| \oplus |x| \qquad x = x \oplus c \Rightarrow |x| \succeq c \qquad |a \oplus b| = |a| \oplus |b|$$

$$x \oplus a = x \oplus b \land |a| = |b| \Rightarrow a = b \qquad \mathsf{stable}(\omega) \Rightarrow \omega = \mathsf{stabilize}(\omega)$$

$$\mathsf{stable}(\mathsf{stabilize}(\omega)) \qquad \mathsf{stabilize}(a \oplus b) = \mathsf{stabilize}(a) \oplus \mathsf{stabilize}(b)$$

$$x = \mathsf{stabilize}(x) \oplus |x| \qquad a = b \oplus \mathsf{stabilize}(|c|) \Rightarrow a = b$$

Figure 2.6.: Axioms for our IDF algebra $(\Sigma, \oplus, |\_|, \mathsf{stable}, \mathsf{stabilize})$. We define $(\omega' \succeq \omega) \triangleq (\exists r.\, \omega' = \omega \oplus r)$.

To capture arbitrary SL and IDF states, we define an *IDF algebra* as follows:

Definition 2.3.1 IDF algebra**.**
*An **IDF algebra** is a quintuple* $(\Sigma, \oplus, |\_|, \mathsf{stable}, \mathsf{stabilize})$ *that satisfies all axioms in Figure 2.6, where* Σ *is a set of states,* $\oplus$ *is a partial, commutative, and associative addition on* Σ (i.e., *a partial function from* $\Sigma \times \Sigma$ *to* Σ*),* $|\_|$ *and* stabilize *are endomorphisms of* Σ*, and* stable *is a predicate on* Σ*.*

The set Σ and the partial addition $\oplus$ are the standard components of a separation algebra. Using $\oplus$, we define the standard partial order $\succeq$ induced by $\oplus$ as $(\omega' \succeq \omega) \triangleq (\exists r.\, \omega' = \omega \oplus r)$. We require *positivity* $(c = a \oplus b = c \land c = c \oplus c \Rightarrow a = a \oplus a)$ to ensure that the partial order is antisymmetric $(a \succeq b \land b \succeq a \Rightarrow a = b)$. Intuitively, the endomorphism $|\_|$ projects a state ω on its largest *duplicable* part, *i.e.*, $|\omega|$ is the largest state smaller than ω such that $|\omega| = |\omega| \oplus |\omega|$. Similarly, the endomorphism stabilize projects a state ω on its largest *stable* part, *i.e.*, $\mathsf{stabilize}(\omega)$ is the largest stable state smaller than ω.

Instantiations. For our concrete IDF state model Σ_{IDF}, the combination $(h_1, \pi_1) \oplus (h_2, \pi_2)$ is defined iff h_1 and h_2 agree on the locations to which both states hold non-zero permission and the sums of their permissions pointwise is at most 1, *i.e.*, iff $\forall l.\, (\pi_1(l) + \pi_2(l) \leq 1) \land (l \in dom(h_1) \cap dom(h_2) \Rightarrow h_1(l) = h_2(l))$. When the combination is defined, $(h_1, \pi_1) \oplus (h_2, \pi_2) \triangleq (h_1 \cup h_2, \pi_1 + \pi_2)$. Knowledge about heap values is duplicable, whereas permissions are not. Thus, $|\_|$ puts all permissions to 0 but preserves the heap, *i.e.*, $|(h, \pi)| \triangleq (h, \lambda l.\, 0)$. Moreover, stabilize erases all values for heap locations to which the state does not hold any permission, *i.e.*, $\mathsf{stabilize}((h, \pi)) \triangleq ((\lambda l.\ \text{if } \pi(l) > 0 \text{ then } h(l) \text{ else } \bot), \pi)$.

Separation algebra instances can also be instantiated as IDF algebras, by defining stable to be true for all states, and stabilize to be the identity function on Σ. For example, Σ_{SL} (defined above) can be instantiated as an IDF algebra with these definitions of stable and stabilize, and with $|\_|$ mapping every state to the unit state (where all permissions are 0, and the domain of the heap is empty). Moreover, like separation algebras [41, 86], IDF algebras support standard constructions like the agreement algebra (where only $\omega = \omega \oplus \omega$ holds), and can be constructed by combining smaller algebras, via combinators such as product and sum types (where both types must be IDF algebras), function types (where only the codomain must be an IDF algebra), etc.

[41]: Jung et al. (2015), *Iris*
[86]: Dockins et al. (2009), *A Fresh Look at Separation Algebras and Share Accounting*

State model for CoreIVL. Our CoreIVL framework can be instantiated for any IDF algebra. We obtain the state model by extending this IDF algebra with a store of local variables, *i.e.*, a partial mapping from variables in *Var* to values in *Val*. Concretely, given an IDF algebra with carrier set Σ, we define the state model for CoreIVL as the product algebra $\Sigma_{IVL} \triangleq ((Var \rightharpoonup Val) \times \Sigma)$, where the store $Var \rightharpoonup Val$ is instantiated to the agreement algebra, *i.e.*, addition on stores is defined for identical stores (as the identity). Using the agreement algebra for the store ensures that `inhale` and `exhale` have no effect on local variables.

Self-framing IDF assertions. Given an arbitrary IDF algebra, we can define a general notion of *self-framing* assertions and a relative notion of assertions *framing* other assertions as follows.

Definition 2.3.2 Properties of IDF assertions**.**
*In the following, P is an IDF assertion (*i.e.*, a set of states from an IDF algebra).*

- P *is* ***self-framing****, written* $\mathsf{selfFraming}(P)$*, iff* $\forall \omega.\ \omega \in P \Leftrightarrow \mathsf{stabilize}(\omega) \in P$.
- *A state* ω ***frames*** P*, written* $\mathsf{frames}(\omega, P)$*, iff* $\mathsf{selfFraming}(\{\omega\} * P)$.
- *An IDF assertion* B ***frames*** P*, written* $\mathsf{frames}(B, P)$*, iff* $\forall \omega \in B.\ \mathsf{stable}(\omega) \Rightarrow \mathsf{frames}(\omega, P)$.
- P ***frames*** *the expression (*i.e.*, a partial function from states to values)* e*, written* $\mathsf{frames}(P, e)$*, iff* $e(\omega)$ *is defined for all states* $\omega \in P$.

Those different notions are tightly connected: If A is self-framing and A frames B then $A * B$ is self-framing. For example, the assertion $A \triangleq (\mathtt{acc(x.f)} * \mathtt{x.f} = 5)$ is self-framing, because any state $\omega_A \in A$ has full permission to `x.f`, and thus $\mathsf{stabilize}(\omega_A)$ will retain the knowledge that `x.f` is 5, and hence $\mathsf{stabilize}(\omega_A) \in A$. In contrast, the assertion $B \triangleq (\mathtt{x.f} = 5)$ is not self-framing, since a state ω_B with no permission to `x.f` but with the knowledge that `x.f` is 5 satisfies B, but $\mathsf{stabilize}(\omega_B)$ will not retain the knowledge that `x.f` = 5, and hence will not satisfy B. Moreover, any state that satisfies `acc(x.f)` frames B, thus the assertion `acc(x.f)` frames B. Note that, in an instantiation with SL states (*e.g.*, Σ_{SL}), all assertions are self-framing, since all SL states are stable.

2.3.2. Operational Semantics

We now formally define the operational semantics of CoreIVL for the state model described above (given an arbitrary IDF algebra). As explained in Section 2.2.3, our operational semantics has judgments of the form $\langle C, \omega \rangle \rightarrow_\Delta S$, where Δ is a type context,[10] C is a statement, ω is a state, and S is a set of states (to capture demonic non-determinism; angelic non-determinism is captured by the existence of different derivations $\langle C, \omega \rangle \rightarrow_\Delta S_1$ and $\langle C, \omega \rangle \rightarrow_\Delta S_2$).

The rules for the operational semantics are given in Figure 2.7. As shown by the rule InhaleOp, `inhale` A can reduce in a state ω only if ω frames A. In our concrete instantiation Σ_{IDF}, this means that ω or A must contain the permission to any heap location mentioned in A. For example, `inhale x.f = 5` can reduce correctly only in a state ω that has some

10: In this chapter, we do not discuss typing in details, but our Isabelle formalization includes it. In particular, it ensures that our operational and axiomatic semantics deal only with well-typed states, *i.e.*, states whose local store and heap contain values of the right types (defined by the type context Δ). By default, all states discussed in this section are well-typed.

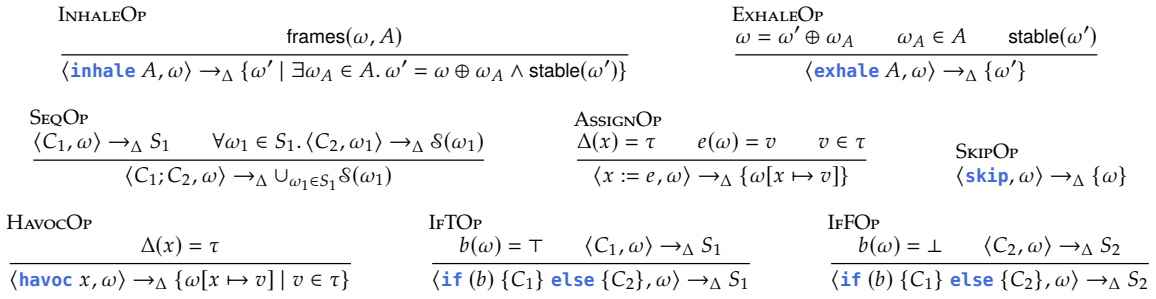

Figure 2.7.: Operational semantics rules.

permission to `x.f`. If ω has a different value than 5 for `x.f`, the statement will reduce to an empty set of states, *i.e.*, $\langle \texttt{inhale x.f = 5}, \omega\rangle \rightarrow_\Delta \emptyset$, capturing the fact that we inhaled an assumption inconsistent with our state. In this case, the rest of the program is trivially correct (because it will be executed in no state). If ω has value 5 for `x.f`, then the statement will reduce to the singleton set $\{\omega\}$, *i.e.*, $\langle \texttt{inhale x.f = 5}, \omega\rangle \rightarrow_\Delta \{\omega\}$. Finally, inhaling **acc**(`x.f`) in a state ω with no permission and no value to `x.f` will result in a set with multiple states (potentially infinitely many), one state for each possible value of `x.f`. We require $\mathsf{stable}(\omega')$ in the rule to ensure that executing a statement in any stable state leads to a set of stable states, *i.e.*, $\mathsf{stable}(\omega) \wedge \langle \omega, C\rangle \rightarrow_\Delta S \Rightarrow (\forall \omega' \in S.\, \mathsf{stable}(\omega'))$. In other words, the operational semantics preserves the stability of states.

Dually, the rule ExhaleOp requires the final state ω' to be stable. This ensures that values of heap locations for which the state lost all permission will be erased. For example, **exhale acc**$(x.f)$ succeeds only in a state with full permission to `x.f`, and results in a final state without any permission nor value for `x.f`. Note that the rule ExhaleOp is the only atomic rule that uses angelic nondeterminism, because the rule can be applied with different ω' (corresponding to different angelic choices). (The rules InhaleOp and HavocOp use demonic non-determinism, while AssignOp and SkipOp are deterministic.) The rule SeqOp first executes C_1 in ω, which yields a set of states S_1. Since S_1 captures demonic choices, C_2 must be executed in *all* states from S_1, but the angelism in C_2 can be resolved differently for each state, which is captured by the choice of the function $\mathcal{S}$. The function $\mathcal{S}$ must map every state ω_1 from S_1 (the set of states obtained after executing C_1 in ω) to a set of states $\mathcal{S}(\omega_1)$ that can be reached by executing C_2 in ω_1.

Finally, note that expressions in CoreIVL are *semantic*, *i.e.*, they are *partial* functions from states to values. We model them as partial functions because they might be heap-dependent, and thus might not be defined for all states. For example, the expression `x.f = 5` is only meaningful in states where `x.f` has a value. The rules AssignOp, IfTOp, and IfFOp require that the expressions are defined in the initial state ω.

2.3.3. Axiomatic Semantics

Using the same extended state model as in the operational semantics, we define an axiomatic semantics with judgments of the form $\Delta \vdash [P]\ C\ [Q]$, where Δ is a type context, P and Q are assertions (sets of states), and C

$$\textsc{SkipAx}\ \frac{\mathsf{selfFraming}(P)}{\Delta \vdash [P]\ \texttt{skip}\ [P]} \qquad \textsc{InhaleAx}\ \frac{\mathsf{selfFraming}(P) \qquad \mathsf{frames}(P, A)}{\Delta \vdash [P]\ \texttt{inhale}\ A\ [P * A]} \qquad \textsc{ExhaleAx}\ \frac{\mathsf{selfFraming}(P) \qquad P \models Q * A \qquad \mathsf{selfFraming}(Q)}{\Delta \vdash [P]\ \texttt{exhale}\ A\ [Q]}$$

$$\textsc{IfAx}\ \frac{\mathsf{selfFraming}(P) \qquad \mathsf{frames}(P, b) \qquad \Delta \vdash [P \wedge b]\ C_1\ [B_1] \qquad \Delta \vdash [P \wedge \neg b]\ C_2\ [B_2]}{\Delta \vdash [P]\ \texttt{if}\ (b)\ \{C_1\}\ \texttt{else}\ \{C_2\}\ [B_1 \vee B_2]} \qquad \textsc{HavocAx}\ \frac{\mathsf{selfFraming}(P) \qquad \Delta(x) = \tau}{\Delta \vdash [P]\ \texttt{havoc}\ x\ [\exists x \in \tau.\, P]}$$

$$\textsc{SeqAx}\ \frac{\Delta \vdash [P]\ C_1\ [R] \qquad \Delta \vdash [R]\ C_2\ [Q]}{\Delta \vdash [P]\ C_1; C_2\ [Q]} \qquad \textsc{AssignAx}\ \frac{\mathsf{selfFraming}(P) \qquad \mathsf{frames}(P, e)}{\Delta \vdash [P]\ x := e\ [\exists v.\, P[v/x] \wedge x = e[v/x]]}$$

Figure 2.8.: Axiomatic semantic rules.

is a CoreIVL statement. All rules are shown in Figure 2.8. Multiple rules have side-conditions requiring the preconditions and postconditions to be self-framing, ensuring that if we have $\Delta \vdash [P]\ C\ [Q]$, P and Q are self-framing.

As explained in Section 2.2.4, our operational and axiomatic semantics are equivalent. The soundness property expressed in Theorem 2.2.1 (in Section 2.2.4) allows one to bridge the gap between a valid CoreIVL program (according to Definition 2.2.1) and the front-end program logic. The proof of Theorem 2.2.1 is not straightforward. In particular, our proof explicitly tracks the angelic choices made based on the sequence of past states of each execution, as shown by the following lemma, which implies Theorem 2.2.1. Let $\ll A \gg \triangleq \{\omega' \mid \mathsf{stabilize}(\omega') \in A\}$ for a set of states A.

Lemma 2.3.1 Correspondence between operational and axiomatic semantics**.**
Given a set $\Omega \in \mathbb{P}(\Sigma^* \times \Sigma)$ of lists of past states paired with current states, a CoreIVL statement C, and a function $\mathcal{S}$ mapping elements from Ω to sets of states, if for all $(l, \omega) \in \Omega$ we have

1. *$\mathsf{stable}(\omega)$, and*
2. *$\langle C, \omega \rangle \rightarrow_\Delta \mathcal{S}(l, \omega)$,*

then $\Delta \vdash [\ll \{\omega \mid (l, \omega) \in \Omega\} \gg]\ C\ [\ll \bigcup_{(l,\omega)\in\Omega} \mathcal{S}(l, \omega) \gg]$.

An element $([\omega_0, \ldots, \omega_n], \omega_{n+1}) \in \Omega$ represents all the intermediate states of one execution up to now, which we use to resolve the future angelism. The function $\mathcal{S}$ maps each such element to a set of states that can be reached from ω_{n+1} by executing C. Intuitively, the precondition collects all the current states from Ω, and the postcondition collects all the states they can reach by executing C. The proof proceeds by structural induction over the statement C.

The reason for tracking sequences of past states. The reader might be wondering why Lemma 2.3.1 keeps track of the list of all past states, instead of only keeping track of the current state. The reason is that only keeping track of the current state would not allow us to prove the inductive case for sequential composition. To understand why, *assume* that Ω is *instead* a set of *single states*, and $\mathcal{S}$ is a function from *single states* to a set of states. Consider proving the inductive case for the sequential composition, *i.e.*, for $C \triangleq (C_1; C_2)$. Assume that, in this scenario, we

are given $\Omega = \{\omega_A, \omega_B\}$, and that $\mathcal{S}$ is such that $\mathcal{S}(\omega_A) = \{\omega'_A\}$ and $\mathcal{S}(\omega_B) = \{\omega'_B\}$. From assumption (2) in Lemma 2.3.1, we know that $\langle C_1; C_2, \omega_A\rangle \to_\Delta \{\omega'_A\}$ and $\langle C_1; C_2, \omega_B\rangle \to_\Delta \{\omega'_B\}$ hold. It might be the case that executing C_1 in either ω_A or ω_B yields the same set of states $\{\omega'\}$, *i.e.*, $\langle C_1, \omega_A\rangle \to_\Delta \{\omega'\}$ and $\langle C_1, \omega_B\rangle \to_\Delta \{\omega'\}$, but that the angelic non-determinism when executing C_2 in state ω' was resolved differently in both executions, leading to ω'_A in the execution from ω_A and ω'_B in the execution from ω_B. More concisely, the executions of $C_1; C_2$ in ω_A and ω_B might have been constructed as follows:

$$\langle C_1, \omega_A\rangle \to_\Delta \{\omega'\} \land \langle C_2, \omega'\rangle \to_\Delta \{\omega'_A\} \Rightarrow \langle C_1; C_2, \omega_A\rangle \to_\Delta \{\omega'_A\}$$
$$\langle C_1, \omega_B\rangle \to_\Delta \{\omega'\} \land \langle C_2, \omega'\rangle \to_\Delta \{\omega'_B\} \Rightarrow \langle C_1; C_2, \omega_B\rangle \to_\Delta \{\omega'_B\}$$

In this case, our intermediate set of states between C_1 and C_2 is $\Omega \triangleq \{\omega'\}$. To apply our induction hypothesis for C_2, we need to find a function $\mathcal{S}_2$ that maps ω' to both $\{\omega'_A\}$ and $\{\omega'_B\}$, as required by the assumption (2) in the lemma, which is not possible.

To solve this issue, we explicitly keep track of all past states. In this way, our intermediate set of states for the previous example is $\Omega \triangleq \{([\omega_A], \omega'), ([\omega_B], \omega')\}$, which allows us to define a function $\mathcal{S}_2$ such that $\mathcal{S}_2([\omega_A], \omega') = \{\omega'_A\}$ and $\mathcal{S}_2([\omega_B], \omega') = \{\omega'_B\}$, allowing us to apply our induction hypothesis and prove the sequential composition case.

Completeness. To show that our operational and axiomatic semantics are equivalent, we also prove the following completeness property (whose proof is less involved than for soundness):

Theorem 2.3.2 Axiomatic-to-operational completeness**.**
Assume $\Delta \vdash [P]\ C\ [Q]$*, and let* $\omega \in P$ *such that* $\mathit{stable}(\omega)$*. Then there exists* S *such that* $\langle C, \omega\rangle \to_\Delta S$ *and* $S \subseteq Q$*.*

2.3.4. ViperCore: Instantiating CoreIVL with Viper

To show the practical usefulness of CoreIVL, we instantiated it for the Viper language. We call this instantiation ViperCore, and we use it in Section 2.4 and Section 2.5. To instantiate the framework presented in this section, one needs (1) an IDF algebra, (2) a type of custom statements C', (3) operational and axiomatic semantic rules for each custom statement, and (4) proofs that those operational and axiomatic semantic rules are compatible with our framework (*i.e.*, soundness and completeness for the custom semantic rules, and a proof that the operational semantics of custom statements preserves stability).

We instantiate (1) with the IDF algebra Σ_{IDF} defined in Section 2.3.1, where the set L of heap locations is the set of pairs of a reference and a field (represented by a string). For (2), we add field assignments as $C' ::= (e_1.f := e_2)$, where e_1 and e_2 are semantic expressions that evaluate to a reference and a value, respectively, and f is a field. The field assignment $e_1.f := e_2$ is deterministic. In an initial state $(\sigma, (h, \pi))$, it reduces to the singleton set $\{(\sigma, (h[(r, f) \mapsto v], \pi))\}$ if e_1 evaluates to a reference r, e_2 evaluates to a value v, and $\pi((r, f)) = 1$. This semantics

is reflected both in its corresponding operational and axiomatic semantic rules (3), and the associated proofs (4) are straightforward.

Moreover, we have also connected the deep embedding of the Viper language developed by Parthasarathy et al. [74] (which we leverage in the next section) to ViperCore, by defining a function $\downarrow C$ that converts their *syntactic* statements, expressions and assertions into *semantic* ViperCore statements, expressions and assertions.

[74]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers*

2.4. Back-End Soundness

In this section, we show how our framework enables formalizing the soundness of different back-end verifiers. We prove the soundness of two fundamentally different verification algorithms commonly used in practice: symbolic execution and verification condition generation. We connect both to the *same* instantiation of CoreIVL, namely ViperCore introduced in Section 2.3.4. This demonstrates that CoreIVL's semantics can accommodate fundamentally different verification algorithms.

Symbolic execution is a common kind of verification algorithm used in separation logic-based verifiers [15, 30, 44]. Section 2.4.1 introduces a symbolic execution back-end for ViperCore. Its design follows Viper's symbolic execution back-end [155], but it is formalized as a function inside Isabelle/HOL. The main result of Section 2.4.1 is a soundness proof of this symbolic execution against the operational semantics of ViperCore, showing how CoreIVL is general enough to justify widely-used symbolic execution algorithms.

[15]: Jacobs et al. (2011), *VeriFast*
[30]: Fragoso Santos et al. (2020), *Gillian, Part i*
[44]: Berdine et al. (2005), *Symbolic Execution with Separation Logic*
[155]: Schwerhoff (2016), *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*

In Section 2.4.2 we connect ViperCore to the formalization by Parthasarathy et al. [74] of Viper's verification condition generation (VCG) back-end, which translates an input Viper program to Boogie.[11] This formalization includes a formal operational semantics of Viper that we call *VCGSem*. Unlike ViperCore, which is designed to capture the verification algorithms of multiple back-ends, VCGSem is specific to the verification algorithm of the VCG back-end. For example, VCGSem uses a total heap (*i.e.*, all possible locations on the heap store a value), while ViperCore is based on a partial heap (which is important to capture existing symbolic execution algorithms). Moreover, VCGSem uses (constrained) *demonic* choice when exhaling wildcard permissions, while ViperCore uses angelic choice. Despite these differences, we show that ViperCore's (and thus also CoreIVL's) operational semantics is general enough to capture VCGSem, which embodies Viper's VCG back-end.

[74]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers*

11: This work provides a proof-producing version of Viper's VCG back-end that generates a certificate in Isabelle for each successful verification, but not a general soundness proof.

We have chosen these two back-ends since they implement very different proof search algorithms: The symbolic execution algorithm manipulates a *symbolic state* including a list of heap chunks, while the VCG back-end maps to Boogie code whose operations are embodied by VCGSem, a big-step operational semantics with a total heap. These back-ends show CoreIVL's generality for justifying multiple common verification algorithms. A key aspect that enables this generality is CoreIVL's use of angelic choice. Concretely, the two back-ends use different algorithms for exhaling wildcard permissions (the symbolic execution halves the permission of one heap chunk while VCGSem *demonically* chooses a

$$\sigma : \mathsf{SymState} := \{\mathsf{store} : \mathsf{Var} \rightharpoonup \mathsf{SymExpr}, \mathsf{pc} : \mathsf{SymExpr}, \mathsf{heap} : \mathsf{List(Chunk)}\}$$
$$t := x \mid l \mid \odot\, t \mid t \oplus t \text{ where } \odot \in \{\neg, -, \ldots\} \text{ and } \oplus \in \{\wedge, \vee, =, +, -, \ldots\}$$
$$c : \mathsf{Chunk} := \{\mathsf{recv} : \mathsf{SymExpr}, \mathsf{field} : \mathsf{FieldName}, \mathsf{perm} : \mathsf{SymExpr}, \mathsf{val} : \mathsf{SymExpr}\}$$

$$\mathsf{sexec}\ \sigma\ C\ K \triangleq \begin{cases} \mathsf{sproduce}\ \sigma\ A\ K & \text{if } C = \texttt{inhale}\ A \\ \mathsf{sconsume}\ \sigma\ A\ (\lambda\sigma.\ .\mathsf{scleanup}\ \sigma\ K) & \text{if } C = \texttt{exhale}\ A \\ \mathsf{sexp}\ \sigma\ e\ (\lambda\sigma.\ t.\mathsf{sexec}\ \mathsf{pc\_add}(\sigma, t)\ C_1\ K & \text{if } C = (\text{if } e \text{ then } C_1 \text{ else } C_2) \\ \quad \wedge\ \mathsf{sexec}\ \mathsf{pc\_add}(\sigma, \neg t)\ C_2\ K) & \\ \ldots & \end{cases}$$

$$\mathsf{sproduce}\ \sigma\ (\mathbf{acc}(e_r.f, e_p))\ K \triangleq \mathsf{sexp}\ \sigma\ e_r\ (\lambda\sigma.\ t_r.\mathsf{sexp}\ \sigma\ e_p\ (\lambda\sigma.\ t_p.\mathsf{chunk\_add}\ \sigma\ \{t_r, f, t_p, \mathsf{fresh}\}\ K))$$
$$\mathsf{sconsume}\ \sigma\ (\mathbf{acc}(e_r.f, \_))\ K \triangleq \mathsf{sexp}\ \sigma\ e_r\ (\lambda\sigma.\ t_r.\mathsf{extract}\ \sigma\ t_r\ f\ \_\ (\lambda\sigma.\ c.\mathsf{chunk\_add}\ \sigma\ c\{\mathsf{perm} := c.\mathsf{perm}/2\}\ K))$$

Figure 2.9.: Symbolic states and excerpts of sexec, sproduce, and sconsume. The full definition can be found in Dardinier et al. [156].

suitably-constrained permission amount). Yet, CoreIVL can capture both algorithms thanks to its use of angelic choice.

2.4.1. Symbolic Execution

We formalized a symbolic execution back-end for ViperCore in Isabelle/HOL based on the description of Viper's back-end by Schwerhoff [155] while also taking inspiration from the (on paper) formalization of symbolic execution by Zimmerman et al. [71].

[155]: Schwerhoff (2016), *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*

[71]: Zimmerman et al. (2024), *Sound Gradual Verification with Symbolic Execution*

Symbolic states. The symbolic state tracked during verification is defined in Figure 2.9. It consists of the following components:[12] (1) A *symbolic store* (store) mapping variables to symbolic expressions, (2) a *path condition* (pc)—which is a symbolic expression tracking logical facts that hold in the current branch of the program—, and (3) a *symbolic heap* (heap) given by a list of heap chunks. *Symbolic expressions* t consist of symbolic variables x, literals l (*e.g.*, for concrete integers, booleans or permission amounts), unary operations $\odot\, t$, and binary operations $t \oplus t$. We define a function $\mathsf{pc\_add}(\sigma, t)$ that adds the (boolean) symbolic expression t to the path condition of σ.

12: For simplicity, we omit components for generating fresh symbolic variables and tracking type information.

The most crucial part of symbolic states is the symbolic heap. As is common [15, 43, 155], we represent the symbolic heap as a list of (heap) chunks. Conceptually, each heap chunk corresponds to an $\mathbf{acc}(e_r.f, e_p)$ resource, which we call an acc-resource in this section, together with an associated value. Concretely, a chunk c is a record with four fields, as shown in Figure 2.9. recv and field describe the heap location that the chunk belongs to, perm describes the permission of the chunk, and val gives the (symbolic) value of the heap location. A symbolic heap is a list of chunks. Note that this list can contain multiple chunks for the same location (*cf.* state consolidation, described shortly).

[15]: Jacobs et al. (2011), *VeriFast*
[43]: Berdine et al. (2005), *Smallfoot*
[155]: Schwerhoff (2016), *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*

Defining the symbolic execution. Our symbolic execution is defined via the sexec function for symbolically executing a statement C. It delegates calls to: the sproduce function (for inhaling an assertion A), the sconsume

function (for exhaling an assertion A), the scleanup function (for removing empty heap chunks after exhaling an assertion), and the sexp function (for symbolically evaluating an expression e). Each of these functions are formalized as functions in Isabelle/HOL and can be executed inside the prover to verify a concrete program. The parts of these functions relevant to this chapter are shown in Figure 2.9. The full definition can be found in Dardinier et al. [156]. Following Schwerhoff [155], these functions are written in continuation passing style with continuation K. This allows us to easily split the verification in multiple branches as shown *e.g.*, by the if-case of sexec. We now highlight the most important aspects of the symbolic execution.

[156]: Dardinier et al. (2025), *Formal Foundations for Translational Separation Logic Verifiers (Extended Version)*

[155]: Schwerhoff (2016), *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*

Representing different state consolidation algorithms. After inhaling an acc-resource and adding it to the list of heap chunks, the symbolic execution might try to merge chunks for the same location and deduce additional information (*e.g.*, that for chunks of the same location their permissions sum to at most 1 and their values match). This process, called *state consolidation* [155], is incorporated into the chunk_add function, used to model inhaling an acc-resource during sproduce:

[155]: Schwerhoff (2016), *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*

$$\mathsf{chunk\_add}\ \sigma\ c\ K \triangleq \mathsf{consolidate}\ \sigma\{\mathsf{heap} := c :: \sigma.\mathsf{heap}\}\ K$$

Since there are many possible ways to implement state consolidation [155] (*e.g.*, merging chunks eagerly or lazily), we do not prescribe a specific implementation of the consolidate function, but instead characterize consolidate *semantically*:[13]

[155]: Schwerhoff (2016), *Advancing Automated, Permission-Based Program Verification Using Symbolic Execution*

13: The actual definition of consolidate is slightly different to decouple the definition of the symbolic execution and ViperCore.

$$\mathsf{consolidate}\ \sigma\ K \triangleq \forall \omega.\ \omega \sim_{\mathsf{sym}} \sigma \Longrightarrow \exists \sigma'.\ \omega \sim_{\mathsf{sym}} \sigma' \wedge K\ \sigma'$$

Concretely, when executing consolidate, one is given a ViperCore state ω related to the current symbolic execution state σ (using the $\sim_{\mathsf{sym}}$ relation) and one can pick an arbitrary new state σ' as long as it is related to the same ViperCore state ω. Intuitively, $\omega \sim_{\mathsf{sym}} \sigma$ is defined by stating that there exists a mapping from symbolic variables to concrete values, which can be simply extended to a mapping from σ to ω. The existential quantifier allows us to represent many different state consolidation algorithms. However, this generality also means that consolidate cannot be executed directly. Instead, one can provide a concrete algorithm and prove it sound against consolidate (our implementation uses the trivial algorithm that does not consolidate at all). However, the soundness proof of our symbolic execution works for any valid consolidation algorithm.

Soundness. We prove sexec sound against the operational semantics of ViperCore:[14]

14: We omit side-conditions about typing to avoid clutter.

Theorem 2.4.1 Soundness of ViperCore's symbolic execution**.**
For each (syntactic) statement C, ViperCore state ω and symbolic state σ related via $\omega \sim_{\mathsf{sym}} \sigma$, if sexec *$\sigma\ C\ K$ evaluates to true, then $\downarrow C$ is correct for the initial state ω.*

$\downarrow C$ is the compilation function from syntactic statements to ViperCore statements described in Section 2.3.4. The operational semantics of CoreIVL is well-suited for this soundness proof since the symbolic

execution also traverses the statements in an operational way, and it is straightforward to relate one ViperCore state to one symbolic execution state via $\omega \sim_{\mathsf{sym}} \sigma$.

Soundness of exhaling wildcards via angelic choice. Let us highlight the most interesting part of this soundness proof: exhaling wildcards. Exhaling assertions is handled by the consume function in Figure 2.9. When exhaling an acc-resource with a wildcard permission amount, consume finds and removes a matching chunk from the symbolic heap using the extract function.[15] Then it adds the chunk back with its permission amount halved. Representing this algorithm directly in ViperCore would be impossible since there might be multiple heap chunks for the same location and thus the amount of permissions removed depends on the structure of the symbolic heap. This structure is not visible in ViperCore, which tracks only a single concrete heap. However, we can still prove this algorithm sound against ViperCore. The angelic choice in the operational semantics allows us to pick *any* non-zero permission amount to remove when constructing the ViperCore execution, in particular, the amount that was chosen by the execution of sexec. This shows how angelic choice gives CoreIVL the flexibility to be used in the soundness proof for different verification algorithms, even some that cannot be represented directly in the CoreIVL.

15: Similar to consolidate, extract is characterized semantically and we provide a default implementation of extract that queries Isabelle/HOL's solvers to find the first matching chunk.

2.4.2. Verification Condition Generation

We now describe how we connect the ViperCore instantiation of CoreIVL to the VCGSem formalization of Viper's VCG introduced by Parthasarathy et al. [74]. VCGSem is expressed as an operational big-step semantics $\langle C, \sigma_t \rangle \rightarrow_{\mathsf{VCG}} r$. Here, C is a (deeply embedded) Viper statement, σ_t the initial VCGSem state consisting of a total heap (mapping all locations to values) and a permission mask (mapping all locations to permission amounts), and r is an *outcome*, which can be either *failure* F, *magic* M, or a *normal outcome* $\mathsf{N}(\sigma_t')$.[16] The key result of Parthasarathy et al. [74] is that for each successful verification run of the VCG algorithm, they provide a proof that the VCGSem execution does not fail: $\neg(\langle C, \sigma_t \rangle \rightarrow_{\mathsf{VCG}} \mathsf{F})$.

[74]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers*

16: We omit typing contexts in this section to avoid clutter.
[74]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers*

What makes the connection between VCGSem and ViperCore interesting is that VCGSem makes various design choices that are specific to the Viper back-end that it was designed to represent. For instance, VCGSem defines the **exhale** of a wildcard to demonically remove a non-zero permission amount smaller than the currently held amount, which precisely mimics Viper's VCG. Morever, VCGSem chooses a total heap representation for the Viper states, where *all* locations store a value (VCGSem checks that only locations with non-zero permission are accessed), because this is how Viper's VCG back-end represents the heap. In contrast, ViperCore uses a more standard partial heap introduced in Section 2.3.1. By proving VCGSem sound against ViperCore, we show that CoreIVL as a general semantics for verification algorithms can capture this preexisting verification algorithm. The most significant challenge in the proof connecting VCGSem and ViperCore is the difference in their heap representations. We explain this challenge and our solutions next.

Total vs. partial heap. The seemingly superficial difference between VCGSem's total heap and ViperCore's partial heap has far-reaching ramifications: a ViperCore execution does not correspond to a *single* VCGSem execution, but rather to a *set* of VCGSem executions.

The reason for this mismatch is in the semantics of `exhale`. When exhaling all permissions to a location and later inhaling permissions to this location again, a Viper semantics needs to pick a *fresh* value for the location such that one cannot unsoundly assume that the value remained unchanged between the `inhale` and the `exhale`. This requirement is naturally expressed with the partial heap of ViperCore: when exhaling all permissions to a location in ViperCore, the location is removed from the partial heap and when new permissions for the location are inhaled, it gets re-added with a (non-deterministically chosen) fresh value. However, since VCGSem uses a total heap, it cannot *remove* locations. Instead, VCGSem non-deterministically assigns these locations new values *after the* `exhale` and leaves the heap unchanged in the `inhale`. Consequently, VCGSem and ViperCore apply (demonic) non-deterministic choice at different program points: VCGSem already picks a fresh value during the `exhale`, while ViperCore chooses it during the `inhale`. To address this mismatch[17], we relate a ViperCore execution not to a single VCGSem execution but to a set of VCGSem executions that represent all possible choices for the demonic non-determinism.

17: The mismatch could also be addressed by changing VCGSem to assign a fresh value during `inhale`. However, our goal is to capture the verification algorithms of *existing* back-ends.

Soundness. We prove the following soundness statement for VCGSem:[18]

18: For readability, we omit some technical assumptions about stability of ω and well-typedness.

Theorem 2.4.2 Soundness of VCGSem.
For all (syntactic) statements C and ViperCore states ω,
if we have $\neg(\langle C, \sigma_t\rangle \rightarrow_{\mathsf{VCG}} F)$ for all VCGSem states σ_t such that $\omega \sim_{\mathsf{VCG}} \sigma_t$,
then $\downarrow C$ is correct for the state ω.

Intuitively, this theorem allows us to transform a proof about a successful verification by the VCG back-end into a verification proof according to the ViperCore semantics. Note that the theorem relates a single ViperCore execution to a *set* of VCGSem executions since the relation $\omega \sim_{\mathsf{VCG}} \sigma_t$ relates a ViperCore state ω to multiple VCGSem states σ_t representing the different choices for the demonic non-determinism. (Otherwise, $\omega \sim_{\mathsf{VCG}} \sigma_t$ is similar to $\omega \sim_{\mathsf{sym}} \sigma$ from Section 2.4.1, but adapted for the different notion of states used by VCGSem.) In fact, to prove Theorem 2.4.2 via induction, we need to prove a stronger lemma that also requires us to construct all possible VCGSem executions for the statement corresponding to the ViperCore execution.

Summary. We have demonstrated in this section how CoreIVL's operational semantics helps us solve Challenge 2, by being general enough to capture the two predominant verification algorithms back-ends implemented in practice: our new formalization of symbolic execution in Section 2.4.1 and the preexisting formalization of Viper's VCG back-end [74] in Section 2.4.2.

[74]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers*

2.5. Front-End Soundness

In this section, we show how our axiomatic semantics addresses Challenge 3 from the introduction, by formalizing and proving sound a concrete front-end translation into ViperCore for a parallel programming language ParImp with loops, shared memory, and dynamic memory allocation and deallocation. We define the language and an IDF-based program logic in Section 2.5.1. In Section 2.5.2, we define the translation of annotated ParImp programs into ViperCore and prove it sound using the axiomatic semantics of ViperCore. While the soundness proof is specific to this translation, it highlights key reusable ingredients and demonstrates how our axiomatic semantics for CoreIVL makes such proofs simple.

2.5.1. An IDF-Based Concurrent Separation Logic

Our parallel programming language ParImp is defined as

$$\begin{aligned} C ::= \quad & x := e \mid x := r.v \mid r.v := e \mid r := \texttt{alloc}(e) \mid \texttt{free}(r) \mid \texttt{skip} \mid \\ & C; C \mid \texttt{if}\ (b)\ \{C\}\ \texttt{else}\ \{C\} \mid C \parallel C \mid \texttt{while}\ (b)\ \{C\} \end{aligned}$$

C ranges over ParImp statements, e over arithmetic expressions, b over boolean expressions, x over integer variables, r over reference variables, and v is a fixed field. We consider objects with a unique field v for simplicity; extending our work to support multiple fields is straightforward. The statement $x := r.v$ loads the value of the field v of the reference r into the variable x, while $r.v := e$ stores the value of the expression e in the field v of the reference r. The statement $r := \texttt{alloc}(e)$ allocates a new reference with the value of the expression e for the field v, and $\texttt{free}(r)$ deallocates the reference r. The other statements are standard. We use a standard small-step semantics $\langle C, \sigma\rangle \rightarrow \langle C', \sigma'\rangle$ where σ and σ' are pairs of a store (a partial mapping from variables to values) and a heap (a partial mapping from pairs of an address and a field to values). The rules of the small-step semantics are given in Appendix A.1.

An IDF-based program logic for ParImp. We build and prove sound a program logic analogous to CSL for ParImp based on our IDF state model Σ_{IDF} (defined in Section 2.3.1). Our framework also supports standard separation logic, but connecting an IDF logic to ViperCore allows us to focus on the most interesting aspects of the soundness proof.

Our program logic judgment is written $\Delta \vdash_{CSL} [P]\ C\ [Q]$, where P and Q are ViperCore assertions (*i.e.*, sets of IDF states). The most important rules of our program logic are given in Figure 2.10. The rules Seq, Cons, If, While, Free, Assign, and Skip, are standard. The rules Alloc, Store, Load, Frame, and Parallel are analogous to the standard CSL rules, but adapted to our IDF setting. In particular, the rule Frame requires the precondition P and the frame F to be self-framing. Without this restriction, one could for example use $P \triangleq (\texttt{acc}(\texttt{r.v}))$ and $F \triangleq (\texttt{r.v} = 5)$ to unsoundly derive the invalid triple $\Delta \vdash_{CSL} [\texttt{acc}(\texttt{r.v}) * \texttt{r.v} = 5]\ \texttt{r.v := 3}\ [(\texttt{acc}(\texttt{r.v}) * \texttt{r.v} = 3) * \texttt{r.v} = 5]$ (whose postcondition is not satisfiable) using the rules Frame and Store. Similarly, the rule Parallel requires the preconditions P_l and P_r

$$\textsc{Frame}\ \frac{\Delta \vdash_{CSL} [P]\ C\ [Q] \qquad \mathsf{selfFraming}(P) \qquad \mathsf{selfFraming}(F) \qquad fv(F) \cap mod(C) = \emptyset}{\Delta \vdash_{CSL} [P * F]\ C\ [Q * F]}$$

$$\textsc{Skip}\ \Delta \vdash_{CSL} [P]\ \texttt{skip}\ [P]$$

$$\textsc{Parallel}\ \frac{mod(C_l) \cap (fv(C_r) \cup fv(Q_r)) = \emptyset \qquad mod(C_r) \cap (fv(C_l) \cup fv(Q_l)) = \emptyset \qquad \Delta \vdash_{CSL} [P_l]\ C_l\ [Q_l] \qquad \Delta \vdash_{CSL} [P_r]\ C_r\ [Q_r] \qquad \mathsf{selfFraming}(P_l) \qquad \mathsf{selfFraming}(P_r)}{\Delta \vdash_{CSL} [P_l * P_r]\ C_l \,||\, C_r\ [Q_l * Q_r]}$$

$$\textsc{Seq}\ \frac{\Delta \vdash_{CSL} [P]\ C_1\ [R] \qquad \Delta \vdash_{CSL} [R]\ C_2\ [Q]}{\Delta \vdash_{CSL} [P]\ C_1; C_2\ [Q]}$$

$$\textsc{Cons}\ \frac{\Delta \vdash_{CSL} [P']\ C\ [Q'] \qquad P \models P' \qquad Q' \models Q}{\Delta \vdash_{CSL} [P]\ C\ [Q]}$$

$$\textsc{If}\ \frac{\Delta \vdash_{CSL} [P \wedge b]\ C_1\ [Q] \qquad \Delta \vdash_{CSL} [P \wedge \neg b]\ C_2\ [Q]}{\Delta \vdash_{CSL} [P]\ \texttt{if}\ (b)\ \{C_1\}\ \texttt{else}\ \{C_2\}\ [Q]}$$

$$\textsc{Alloc}\ \frac{r \notin fv(e)}{\Delta \vdash_{CSL} [\top]\ \texttt{r := alloc(e)}\ [\texttt{acc(r.v)} * \texttt{r.v = e}]}$$

$$\textsc{While}\ \frac{\Delta \vdash_{CSL} [I \wedge b]\ C\ [I]}{\Delta \vdash_{CSL} [I]\ \texttt{while}\ (b)\ \{C\}\ [I \wedge \neg b]}$$

$$\textsc{Load}\ \frac{P \models \texttt{acc(r.v, \_)}}{\Delta \vdash_{CSL} [P]\ \texttt{x := r.v}\ [\exists u.\, P[u/x] * \texttt{x = r.v}]}$$

$$\textsc{Store}\ \Delta \vdash_{CSL} [\texttt{acc(r.v)}]\ \texttt{r.v := e}\ [\texttt{acc(r.v)} * \texttt{r.v = e}]$$

$$\textsc{Free}\ \Delta \vdash_{CSL} [\texttt{acc(q.v)}]\ \texttt{free(q)}\ [\top]$$

$$\textsc{Assign}\ \Delta \vdash_{CSL} [P[x/e]]\ \texttt{x := e}\ [P]$$

Figure 2.10.: Inference rules of our IDF-based concurrent separation logic.

to be self-framing. Finally, the rule Load allows arbitrary preconditions P, as long as P asserts some permission to read `r.v`.

Soundness of the IDF-based program logic. To prove the soundness and the adequacy of this program logic, we adapt Vafeiadis [157]'s approach to our IDF setting. We first define a predicate $safe_n(C, s, \omega, Q)$ where n is a natural number, C is a ParImp command, s is a store, $\omega \in \Sigma_{IDF}$ is an IDF state, and $Q \subseteq \Sigma_{IVL}$ is a set of IDF states with stores of local variables. Intuitively, $safe_n(C, s, \omega, Q)$ holds iff it is safe to execute the command C for n steps in an initial state corresponding to the IDF state $(s, \omega) \in \Sigma_{IVL}$, and any final state (reached within n steps) will correspond to an IDF state satisfying the postcondition Q. A CSL triple $\Delta \vdash_{CSL} [P]\ C\ [Q]$ is then valid, written $\Delta \models_{CSL} [P]\ C\ [Q]$, iff $safe_n(C, s, \omega, Q)$ holds for all n and for all states $(s, \omega) \in P$ such that ω is stable.

[157]: Vafeiadis (2011), *Concurrent Separation Logic and Operational Semantics*

Definition 2.5.1 Validity of CSL triples.
We denote the sets of heap locations accessed and modified by C in one step as accesses(C, s) and writes(C, s), respectively, which we formally define in Appendix A.1. Moreover, we define the functions readDom(ω) and writeDom(ω) as the sets of locations for which ω has reading and writing permissions respectively, i.e., *readDom$(\omega) \triangleq \{l \mid \exists v, p.\, \omega(l) = (v, p) \wedge p > 0\}$ and writeDom$(\omega) \triangleq \{l \mid \exists v.\, \omega(l) = (v, 1)\}$.*

Moreover, an IDF state $\omega \in \Sigma_{IDF}$ corresponds to a heap h (a partial map from heap locations to values), written $\omega \rightsquigarrow h$, iff they have the same domain and agree on their values, and ω has exclusive permission to all locations in its domain:

$$\omega \rightsquigarrow h \triangleq (dom(\omega) = dom(h) \wedge (\forall l \in dom(\omega).\, \omega(l) = (h(l), 1)))$$

We can now define the predicate $safe_n(C, s, \omega, Q)$*, where* n *is a natural number,* C *a ParImp command,* s *a store,* $\omega \in \Sigma_{IDF}$ *an IDF state, and* $Q \subseteq \Sigma_{IVL}$ *a set of IDF states with stores, as follows:*

$safe_0(C, s, \omega, Q)$ *always holds.*

$safe_{n+1}(C, s, \omega, Q)$ *holds iff the following conditions hold:*[19]

1. *If* $C = \textbf{skip}$ *then* $(s, \omega) \in Q$
2. $accesses(C, s) \subseteq readDom(\omega)$ *and* $writes(C, s) \subseteq writeDom(\omega)$
3. *For all heaps* h *and IDF states* ω_f*, if* $\omega \# \omega_f$*,* $\omega \oplus \omega_f \rightsquigarrow h$*, and* $stable(\omega_f)$*, then* $\langle C, (s, h)\rangle \nrightarrow \bot$
4. *For all heaps* h*, IDF states* ω_f*, commands* C'*, stores* s'*, and heaps* h'*, if* $\omega \# \omega_f$*,* $\omega \oplus \omega_f \rightsquigarrow h$*,* $stable(\omega_f)$*, and* $\langle C, (s, h)\rangle \rightarrow \langle C', (s', h')\rangle$*, then there exists another state* ω' *such that* $\omega' \# \omega_f$*,* $\omega' \oplus \omega_f \rightsquigarrow h'$*,* $stable(\omega')$*, and* $safe_n(C', s', \omega', Q)$*.*

Finally, we define the validity of CSL triples as follows:

$$\Delta \models_{CSL} [P]\ C\ [Q] \triangleq \big(\forall (s, \omega) \in P.\ stable(\omega) \Rightarrow (\forall n.\ safe_n(C, s, \omega, Q))\big)$$

19: Ignoring typing to avoid clutter.

In the definition of $safe_{n+1}(C, s, \omega, Q)$, condition 1 ensures that final states satisfy the postcondition, conditions 2 and 3 ensure the absence of data races and crashes, and condition 4 performs a step and ensures that this safety predicate holds for the next n steps as well. In conditions 3 and 4, ω_f represents the *frame*, which is intuitively the (fractional) part of the *global* state of the program that is not owned by the current *local* state. The parts specific to our IDF setting are highlighted in blue: All IDF states considered in this definition (ω, ω', and ω_f) must be stable (which is why the rules Frame and Parallel require the assertions in the precondition to be self-framing). In other words, unstable states are used only internally to give a semantics to IDF assertions, but they do not matter when giving a semantics to CSL triples.[20]

20: In Chapter 3, we will introduce a notion of *unbounded* states that play a similar role.

We have proven in Isabelle that this definition of validity for CSL triples is adequate, in the following sense:

Theorem 2.5.1 Adequacy of the CSL triples**.**
Let C *be a well-typed program, and* P *and* Q *be predicates on ParImp states (*i.e.*, without permissions). If the triple* $\Delta \models_{CSL} [P]\ C\ [Q]$[21] *holds, and if* σ *is a well-typed state such that* $P(\sigma)$*, then executing* C *in the state* σ *will not abort nor encounter any data race, and for all* σ' *such that* $\langle C, \sigma\rangle \rightarrow^* \langle \texttt{skip}, \sigma'\rangle$*, we have* $Q(\sigma')$*.*

21: Here, we abuse the notation and assume that P and Q are extended to IDF states in the obvious way.

Finally, we have proven in Isabelle that the rules from Figure 2.10 are *sound*, in the following sense:[22]

22: Still ignoring typing.

Theorem 2.5.2 Soundness of the IDF-based CSL**.**
If $\Delta \vdash_{CSL} [P]\ C\ [Q]$ *holds, then* $\Delta \models_{CSL} [P]\ C\ [Q]$ *holds.*

2.5.2. A Sound Front-End Translation

Building on the previously-defined IDF-based program logic, we define a standard front-end translation from ParImp programs with annotations into ViperCore programs, shown in Figure 2.11. This translation was

$$
\begin{array}{lll}
[\![r := \texttt{alloc}(e)]\!] & \triangleq ((\texttt{havoc}\ r; \texttt{inhale}\ \texttt{acc}(r.v) * r.v = e), \varnothing) & [\![\texttt{skip}]\!] \triangleq (\texttt{skip}, \varnothing) \\
[\![\texttt{free}(r)]\!] & \triangleq (\texttt{exhale}\ \texttt{acc}(r.v), \varnothing) & [\![x := e]\!] \triangleq (x := e, \varnothing) \\
[\![C_1; C_2]\!] & \triangleq (([\![C_1]\!].1; [\![C_2]\!].1), ([\![C_1]\!].2 \cup [\![C_2]\!].2)) & [\![r.v := e]\!] \triangleq (r.v := e, \varnothing) \\
[\![\texttt{if}\ (b)\ \{C_1\}\ \texttt{else}\ \{C_2\}]\!] & \triangleq (\texttt{if}\ (b)\ \{[\![C_1]\!].1\}\ \texttt{else}\ \{[\![C_2]\!].1\}), ([\![C_1]\!].2 \cup [\![C_2]\!].2)) & [\![x := r.v]\!] \triangleq (x := r.v, \varnothing)
\end{array}
$$

$$
\begin{array}{ll}
[\![C_l \mid\mid C_r]\!] & \triangleq ((\texttt{exhale}\ P_l * P_r; \texttt{havoc}\ \mathit{mod}(C_l) \cup \mathit{mod}(C_r); \texttt{inhale}\ Q_l * Q_r), \\
 & \quad \{\texttt{inhale}\ P_l; [\![C_l]\!].1; \texttt{exhale}\ Q_l\} \cup \{\texttt{inhale}\ P_r; [\![C_r]\!].1; \texttt{exhale}\ Q_r\} \cup [\![C_l]\!].2 \cup [\![C_r]\!].2) \\
[\![\texttt{while}\ (b)\ \{C\}]\!] & \triangleq ((\texttt{exhale}\ I; \texttt{havoc}\ \mathit{mod}(C); \texttt{inhale}\ I \wedge \neg b), \\
 & \quad \{\texttt{inhale}\ I \wedge b; [\![C]\!].1; \texttt{exhale}\ I\} \cup [\![C]\!].2)
\end{array}
$$

Figure 2.11.: Front-end translation from ParImp to ViperCore. The translation function $[\![\_]\!]$ takes as input an annotated ParImp statement C and returns a pair of a ViperCore statement and a set of ViperCore statements. We write $[\![C]\!].1$ and $[\![C]\!].2$ to denote its first and second components, respectively. Assertions P_l, P_r, Q_l, and Q_r for the parallel composition and I for the while loop are annotations provided by the user, which are all required to be self-framing. The notation **havoc** V, where V is a set of variables $\{x_1, \ldots, x_n\}$, is a shorthand for **havoc** $x_1; \ldots;$ **havoc** x_n.

illustrated in the example in Figure 2.3 from Section 2.2. The translation function $[\![\_]\!]$ takes as input an annotated ParImp statement C and yields a *pair of a ViperCore statement and a set of ViperCore statements*. The first component, written $[\![C]\!].1$, corresponds to the main translation of C, while the second component, written $[\![C]\!].2$, corresponds to the set of auxiliary Viper methods generated by the translation along the way. Auxiliary methods are generated for loops and parallel compositions only. Methods `l` and `r` in Figure 2.3 are examples of such auxiliary methods.

The translation of field and variable assignments is straightforward. The translation of sequential composition and conditional statements is also straightforward since they use the corresponding sequential composition and conditional statements of ViperCore, and collect the auxiliary methods generated by the translation of the sub-statements. The translation of allocation and deallocation statements corresponds to the rules Alloc and Free from Figure 2.10.

The translation of parallel composition and while loops is more involved, but they follow the same pattern. First, the premises of the relevant rules (Parallel and While) are checked by generating auxiliary methods, which first inhale the relevant precondition, then translate the relevant statement, and finally exhale the relevant postcondition. For example, the premise $\Delta \vdash_{CSL} [I \wedge b]\ C\ [I]$ of the rule While is checked by generating the auxiliary method **inhale** $I \wedge b; [\![C]\!].1;$ **exhale** I. We call this pattern the *inhale-translation-exhale* pattern. Then, the main translation follows the conclusion of the rule, by exhaling the precondition, havocking the modified variables, and inhaling the postcondition. For example, the main translation of the loop **while** $(b)\ \{C\}$ is **exhale** I; **havoc** $\mathit{mod}(C)$; **inhale** $I \wedge \neg b$, reflecting the conclusion $\Delta \vdash_{CSL} [I]$ **while** $(b)\ \{C\}\ [I \wedge \neg b]$ of the rule *While*. We call this pattern, which we have already seen in Section 2.2.4, the *exhale-havoc-inhale* pattern. Those two patterns are not specific to our translation, but are general patterns that can be found in many front-end translations.

Soundness. We assume that the ParImp statement C we want to verify is annotated with a precondition P and a postcondition Q. In this case, we add **inhale** P before the main translation (as we did in Figure 2.3), and **exhale** Q afterwards, following the inhale-translation-exhale pattern.

Our complete front-end translation yields the set of ViperCore statements $\{\texttt{inhale}\ P; [\![C]\!].1; \texttt{exhale}\ Q\} \cup [\![C]\!].2$. Our translation is sound, as stated in the following theorem. We say that a ViperCore statement C_v is valid *w.r.t. the axiomatic semantics*, which we write $valid_{Ax}(C_v)$, iff $\exists B.\, \Delta \vdash [\top]\ C_V\ [B]$.

Theorem 2.5.3 Soundness of the front-end translation**.**
Let C be a front-end statement, and P and Q be assertions. If
(1) the axiomatic semantics triple $\Delta \vdash [P]\ [\![C]\!].1\ [Q]$ holds, and
(2) all ViperCore statements in $[\![C]\!].2$ are valid w.r.t. the axiomatic semantics,
then $\Delta \vdash_{CSL} [P]\ C\ [Q]$ holds.

To prove this theorem, we show that the translation of every front-end statement C is *backward-convertible* (or *convertible* in short), which we write as *convertible*(C). Intuitively, this means that if the translation of the front-end statement into ViperCore is valid (including all auxiliary ViperCore methods) then we can convert the axiomatic semantics triple $\Delta \vdash_{CSL} [P]\ [\![C]\!].1\ [Q]$ into a front-end triple $\Delta \vdash_{CSL} [P]\ C\ [Q]$. Formally, *convertible*(C) holds iff

$$\forall P, Q.\, ((\forall C_v \in [\![C]\!].2.\, valid_{Ax}(C_v)) \wedge \Delta \vdash [P]\ [\![C]\!].1\ [Q]) \Rightarrow \Delta \vdash_{CSL} [P]\ C\ [Q]$$

This convertibility property combined with the following lemma allows us to prove Theorem 2.5.3:

Lemma 2.5.4 Inhale-translation-exhale pattern**.**
If
(1) all auxiliary methods from $[\![C]\!].2$ are valid w.r.t. the axiomatic semantics,
(2) convertible(C) holds, and
(3) $\Delta \vdash [P]\ \texttt{inhale}\ A; [\![C]\!].1; \texttt{exhale}\ B\ [Q]$ holds,
then $\Delta \vdash_{CSL} [P * A]\ C\ [B * Q]$ holds.

Proof. By inverting the rules SeqAx, InhaleAx, and ExhaleAx in (3), we get the existence of R such that (a) $\Delta \vdash [P * A]\ [\![C]\!].1\ [R]$ holds and (b) $R \models B * Q$. By applying *convertible*(C) (from (2)), and from (1) and (a), we get $\Delta \vdash_{CSL} [P * A]\ C\ [R]$. We conclude by combining (b) with the rule Cons. □

The proof of this lemma is straightforward thanks to CoreIVL's *axiomatic* semantics. Relating CSL to an operational IVL semantics would require substantially more effort to re-prove standard reasoning principles, which we prove once and for all in the equivalence proof of the two IVL semantics.

We now need to prove *convertible*(C) for all C, which we do by structural induction. The inductive cases for most statements are straightforward; the interesting cases are allocation, deallocation, parallel compositions, and while loops. As explained above, the main translation of those statements follows the same exhale-havoc-inhale pattern, which we have already seen in Section 2.2.4, and prove below:

Lemma 2.5.5 Exhale-havoc-inhale pattern**.**

Let P and Q be self-framing assertions[23]*, and $\mathcal{L}$ be a separation logic with a frame rule and a consequence rule (such as our IDF-based CSL).*
If $\Delta \vdash [A]$ **exhale** P;**havoc** x_1;...;**havoc** x_n;**inhale** Q $[B]$ *holds, where* $\{x_1, \ldots, x_n\} = mod(C)$*, then* $\Delta \vdash_{\mathcal{L}} [A]\ C\ [B]$ *holds.*

23: This condition is trivially true for standard SLs.

Proof. By inverting the rule SeqAx, we obtain (a) $\Delta \vdash [F]$ **inhale** Q $[B]$ and (b) $\Delta \vdash [A]$ **exhale** P;**havoc** x_1;...;**havoc** x_n $[F]$ for some assertion F. From (b), by inverting the rules SeqAx and HavocAx, we obtain an assertion R such that (c) $\Delta \vdash [A]$ **exhale** P $[R]$ holds, (d) $fv(F) \cap \{x_1, \ldots, x_n\} = \emptyset$, and (e) $R \models F$.[24] By applying the frame rule with F and $\Delta \vdash_{\mathcal{L}} [P]\ C\ [Q]$, where the side condition is justified by (d), we get $\Delta \vdash_{\mathcal{L}} [P * F]\ C\ [Q * F]$. Finally, we obtain $B = F * Q$ from (a) (by inverting the rule InhaleAx), and $A \models P * F$ from (c) (by inverting the rule ExhaleAx) and (e); applying the consequence rule yields $\Delta \vdash_{\mathcal{L}} [A]\ C\ [B]$. □

24: More precisely, we obtain $F = (\exists x_1, \ldots, x_n.\ R)$, from which (d) and (e) follow.

This proof shows that, in this pattern, the role of the **exhale** statement, followed by a sequence of **havoc** statements, is to compute (implicity) the suitable frame for the front-end statement. The **inhale** statement afterwards then adds the postcondition of the front-end statement to the frame.

convertible(**free**(r)) and *convertible*(r := **alloc**(e)) follow directly from the lemma above, by observing that **inhale** $\top$ and **exhale** $\top$ are equivalent to **skip** (and so omitted when encoding).

To prove *convertible*(**while** (b) $\{C\}$) (assuming C is convertible), we first apply Lemma 2.5.4 on the auxiliary method **inhale** $I \wedge b$; $[\![C]\!]$.1; **exhale** I to get $\Delta \vdash_{CSL} [I \wedge b]\ C\ [I]$. We then apply the rule While to get $\Delta \vdash_{CSL} [I]$ **while** (b) $\{C\}$ $[I \wedge \neg b]$. Finally, we conclude by applying Lemma 2.5.5 on the main translation (**exhale** I; **havoc** $mod(C)$; **inhale** $I \wedge \neg b$).

The proof of *convertible*($C_1 || C_2$) proceeds similarly, by first applying Lemma 2.5.4 on the two auxiliary methods (corresponding to the two premises of the rule Parallel), then applying the rule Parallel, and concluding by applying Lemma 2.5.5. This concludes the proof of *convertible*(C) for all C, and thus the proof of Theorem 2.5.3.

Summary. We have demonstrated how the axiomatic semantics from Section 2.3.3 helps us solve Challenge 3, by allowing us to prove general lemmas about patterns that are common in front-end translations in a simple and straightforward manner, and to prove the soundness of a concrete front-end translation for a parallel programming language with multiple features not present in the IVL (*e.g.*, loops, dynamic memory allocation and deallocation).

2.6. Related Work

Semantics of SL-based IVLs. There are two recent formalizations [71, 74] of subsets of Viper [16]. However, each of them exposes implementation details of a Viper back-end, which does not allow the semantics to be connected to diverse back-ends and also not easily to front-ends. In particular, in work not presented in this thesis [74], we use a total heap representation reflecting the Viper VCG back-end that translates to

[71]: Zimmerman et al. (2024), *Sound Gradual Verification with Symbolic Execution*
[74]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers*
[16]: Müller et al. (2016), *Viper*
[74]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers*

Boogie (as discussed in Section 2.4.2), and Zimmerman et al. [71] reflect Viper's symbolic execution back-end.

[71]: Zimmerman et al. (2024), *Sound Gradual Verification with Symbolic Execution*

GIL [68], which is the intermediate language of Gillian [30, 63], is parametric in its (1) state model, which must be provided as a PCM (supporting SL but not IDF states in contrast to CoreIVL), (2) *memory actions* operating on the state model, and (3) *core predicates* describing atomic assertions on the memory such as a SL points-to assertion. For each state instantiation, tool developers targeting GIL must specify *produce* and *consume* actions for each core predicate, which correspond to `inhale` and `exhale` operations in CoreIVL. Together with instantiated parameters, Fragoso Santos et al. [30] provide an operational semantics for the symbolic execution of GIL. Since the instantiated state effectively reflects the symbolic state on which the symbolic execution tool operates, a GIL instantiation essentially represents the back-end semantics. This is in contrast to our CoreIVL, which allows abstracting over multiple back-ends.

[68]: Maksimović et al. (2021), *Gillian*
[30]: Fragoso Santos et al. (2020), *Gillian, Part i*
[63]: Maksimović et al. (2021), *Gillian, Part II*
[30]: Fragoso Santos et al. (2020), *Gillian, Part i*

In work not presented in this thesis [73], we define an alternative semantics of a parametric verification language similar to CoreIVL for the purpose of showing formal results on method call inlining and loop unrolling in automated SL verifiers. This alternative semantics is meant to capture IVL *back-ends* with their heuristics. That is, an instantiation reflects a *single* back-end. As a result, in contrast to CoreIVL, this alternative semantics has no angelic nondeterminism. Moreover, the notion of separation algebra used in this work to represent states does not support IDF.

[73]: Dardinier et al. (2023), *Verification-Preserving Inlining in Automatic Separation Logic Verifiers*

Proofs connecting a front-end with an IVL. Summers and Müller [158] and Wolf et al. [159] reason about the correctness of translations into a SL-based IVL by providing proof sketches for mapping a correct Viper program to a proof for Hoare triples in the RSL weak memory logic [160] and the TaDa logic [161], respectively. However, the reasoning is done 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. Neither of these works formally reasons about the underlying Viper semantics; they describe the behavior of Viper encodings informally.

[158]: Summers et al. (2020), *Automating Deductive Verification for Weak-Memory Programs (Extended Version)*
[159]: Wolf et al. (2022), *Concise Outlines for a Complex Logic*
[160]: Vafeiadis et al. (2013), *Relaxed Separation Logic*
[161]: da Rocha Pinto et al. (2014), *TaDA*

Maksimović et al. [68] briefly describe a parametric soundness framework for GIL (the intermediate language of Gillian [30, 63]). They show that if certain conditions hold on the instantiations of the GIL parameters, then the resulting symbolic execution is sound w.r.t. a concretization function on symbolic states. However, they do not provide an IVL semantics like CoreIVL that abstracts uniformly over multiple back-ends. Additionally, since GIL does not support concurrency [30, 63], their soundness framework cannot reason about the encoding of front-end languages such as ParImp described in Section 2.5. Lööw et al. [69] present a formal compositional symbolic execution engine inspired by Gillian. In contrast to our work, they focus on supporting both over-approximating and under-approximating reasoning, and do not model an IVL, but only apply their framework to a simple front-end language with a fixed memory model.

[68]: Maksimović et al. (2021), *Gillian*
[30]: Fragoso Santos et al. (2020), *Gillian, Part i*
[63]: Maksimović et al. (2021), *Gillian, Part II*
[30]: Fragoso Santos et al. (2020), *Gillian, Part i*
[63]: Maksimović et al. (2021), *Gillian, Part II*
[69]: Lööw et al. (2024), *Compositional Symbolic Execution for Correctness and Incorrectness Reasoning*

Interestingly, [162] generate SL proofs for automatically synthesized heap-manipulating programs, by converting synthesis proof trees into

[162]: Watanabe et al. (2021), *Certifying the Synthesis of Heap-Manipulating Programs*

verification proof trees, which is analogous to how we convert proof trees from CoreIVL's axiomatic semantics to the front-end's separation logic in Section 2.5.

There is also work proving the soundness of front-end translations to IVLs not based on SL [74, 77, 163–165]. However, in contrast to our setting, the corresponding translations do not reflect rules in a front-end program logic. As a result, the soundness proofs work naturally at the level of an operational semantics for the front-end and IVL. Examples include translations from the Dminor data processing language to the Bemol IVL [164], from C to the WhyCert IVL (inspired by the Why3 IVL) [77], and from Viper to Boogie [74] (in the case of the Viper-to-Boogie translation, Viper is the front-end and Boogie is the target IVL).

[74]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers*
[77]: Herms (2013), *Certification of a Tool Chain for Deductive Program Verification*
[163]: Vogels et al. (2009), *A Machine Checked Soundness Proof for an Intermediate Verification Language*
[164]: Backes et al. (2011), *Automatically Verifying Typing Constraints for a Data Processing Language*
[165]: Fortin (2013), *BSP-Why, Un Outil Pour La Vérification Déductive de Programmes BSP*

Proofs connecting an IVL with a back-end. In work not presented in this thesis [74], we show the soundness of the Viper back-end that translates to Boogie. In this chapter, we show that the back-end specific semantics from this other work respects our more generic version (Section 2.4.2). The work most closely related to the symbolic execution back-end presented in Section 2.4.1 is Zimmerman et al. [71]'s formalization of a variant of Viper's symbolic execution back-end targeted at gradual verification. Due to their focus on gradual verification, they only target a simplified model of Viper that (unlike our symbolic execution) does not support fractional permissions. As a consequence, they can use a simpler implementation that does not rely on continuation passing style and they can ignore some of the complexities described in Section 2.4.1 such as state consolidation. Moreover, they formalize the symbolic execution via a derivation tree, while we implement it as an Isabelle/HOL function. Jacobs et al. [67] prove a formalization of VeriFast's symbolic execution sound. Compared to our work, they do not have a semantics that captures different verification algorithms, or supports IDF or fractional permissions.

[164]: Backes et al. (2011), *Automatically Verifying Typing Constraints for a Data Processing Language*
[77]: Herms (2013), *Certification of a Tool Chain for Deductive Program Verification*
[74]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers*
[74]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers*
[71]: Zimmerman et al. (2024), *Sound Gradual Verification with Symbolic Execution*
[67]: Jacobs et al. (2015), *Featherweight VeriFast*

There is also work on non SL-based IVL back-end proofs. These back-ends typically have simple state models and use different algorithms compared to SL-based back-ends. For example, Parthasarathy et al. [75] generate soundness proofs for Boogie's VCG, and Vogels et al. [166] prove a VCG for a similar IVL sound once and for all. Garchery [167] and Cohen and Johnson-Freyd [76] validate certain logical transformations performed in the Why3 IVL verifier.

[75]: Parthasarathy et al. (2021), *Formally Validating a Practical Verification Condition Generator*
[166]: Vogels et al. (2010), *A Machine-Checked Soundness Proof for an Efficient Verification Condition Generator*
[167]: Garchery (2021), *A Framework for Proof-carrying Logical Transformations*
[76]: Cohen et al. (2024), *A Formalization of Core Why3 in Coq*

Angelic non-determinism. Angelic non-determinism [168] has been widely used from encoding partial programs [169], to representing interaction between code written in multiple languages [154, 170], to encoding specifications [168, 171]. However, to the best of our knowledge, our work is the first to use angelic non-determinism to abstract over different verification algorithms. Jacobs et al. [67] and Song et al. [171] both also use angelism for `exhale`, but do not abstract over or formally connect with diverse back-end algorithms, as we do. Instead, Jacobs et al. [67] use angelism to represent a symbolic execution algorithm, while Song et al. [171] use angelism to encode the transfer of resources in a refinement calculus.

[168]: Floyd (1967), *Nondeterministic Algorithms*
[169]: Bodik et al. (2010), *Programming with Angelic Nondeterminism*
[154]: Guéneau et al. (2023), *Melocoton*
[170]: Sammler et al. (2023), *DimSum*
[168]: Floyd (1967), *Nondeterministic Algorithms*
[171]: Song et al. (2023), *Conditional Contextual Refinement*
[67]: Jacobs et al. (2015), *Featherweight VeriFast*
[171]: Song et al. (2023), *Conditional Contextual Refinement*
[67]: Jacobs et al. (2015), *Featherweight VeriFast*
[171]: Song et al. (2023), *Conditional Contextual Refinement*

Implicit dynamic frames (IDF). IDF was originally presented with a fixed resource model (*i.e.*, full ownership to a heap location) and where the heap is represented as a *total* mapping from heap locations to values [70]. Parkinson and Summers [87] formally showed the relationship between IDF and SL by defining a logic based on *total* heaps and separate permission masks that captures both. They also consider fixed resource models of IDF and SL (*i.e.*, fractional ownership to a heap location [81]). Our work generalizes the notion of a separation algebra [85, 86] to capture arbitrary resource models for IDF *and* SL in the same framework. In particular, the algebra does not fix a particular state representation. This enables, for instance, a *partial* heap instantiation for IDF that we use to formalize Viper's state model (Section 2.3.4).

[70]: Smans et al. (2012), *Implicit Dynamic Frames*
[87]: Parkinson et al. (2012), *The Relationship Between Separation Logic and Implicit Dynamic Frames*
[81]: Boyland (2003), *Checking Interference with Fractional Permissions*
[85]: Calcagno et al. (2007), *Local Action and Abstract Separation Logic*
[86]: Dockins et al. (2009), *A Fresh Look at Separation Algebras and Share Accounting*

In work published after this chapter, Spies et al. [172] integrate IDF into Iris [31, 41], a framework for higher-order concurrent SL. To do so, they extend the notion of *resource algebras*, at the core of Iris, to support *unstable resources*, similar to how our novel notion of IDF algebra extends the classical notion of separation algebra with unstable states. They also add a notion of *unstable core*, which corresponds to our operator |_|.

[172]: Spies et al. (2025), *Destabilizing Iris*
[31]: Jung et al. (2018), *Iris from the Ground Up*
[41]: Jung et al. (2015), *Iris*

SteelCore [49] is a framework with an extensible CSL to reason about concurrent F* [14] programs. The extensibility of the framework is in particular demonstrated by allowing IDF-style preconditions of the restricted form $P * b$ (compared to the more general IDF assertions supported in our work), where P is an SL assertion, and b is a heap-dependent boolean expression framed by P (and similarly for postconditions).

[49]: Swamy et al. (2020), *SteelCore*
[14]: Swamy et al. (2016), *Dependent Types and Multi-Monadic Effects in F**

Other approaches. In this chapter, we showed how one can formally establish the soundness of translational SL verifiers, but there are other approaches to building automated SL verifiers and establishing their soundness, as discussed in Chapter 1. Steel [48] is an SL-based proof-oriented programming language in F*. Steel programs are automatically proved correct using a type checker that is proved sound against SteelCore; the type checker uses an SMT solver to discharge proof obligations.

[48]: Fromherz et al. (2021), *Steel*

Keuchel et al. [47] (building on the ideas of Jacobs et al. [67]) show how to build a verified symbolic execution based on a specification monad that allows expressing angelic and demonic non-determinism and assume and assert statements. They formalize (in Coq) two (structurally identical) versions of the symbolic execution algorithm: a deeply embedded version that allows execution and a shallow embedded version to prove soundness of the former. However, both versions represent the same algorithm; they do not consider different back-ends (like the verification condition generation back-end in Section 2.4.2).

[47]: Keuchel et al. (2022), *Verified Symbolic Execution with Kripke Specification Monads (and No Meta-Programming)*
[67]: Jacobs et al. (2015), *Featherweight VeriFast*

Sammler et al. [37] propose an approach to building sound verifiers that requires writing the verifier in a domain specific language called Lithium. Verifiers in Lithium can be automatically executed inside the Coq proof assistant and produce a foundational proof of correctness. Lithium-based verifiers are not translational, but work directly on the source-language program.

[37]: Sammler et al. (2021), *RefinedC*

3. Fractional Predicates

Two plus two is four
Minus one that's three, quick maths

Big Shaq, *Man's Not Hot*

The previous chapter introduced a general formal framework for establishing the soundness of translational verifiers based on separation logic (SL). At the core of this framework is CoreIVL, a generic language that can be instantiated with various SL-based intermediate verification languages. CoreIVL operates at the level of semantic assertions, that is, functions from states to Booleans.

In this chapter, we focus on the semantics of SL assertions themselves. We present a semantics for a language of SL assertions as supported by automated verifiers, encompassing key features such as magic wands and fractional predicates. Notably, we highlight a fundamental mismatch between the treatment of fractional predicates in theoretical work and their implementation in automated verifiers. To bridge this gap, we introduce a novel semantics for SL assertions, called *unbounded separation logic*, which permits states to temporarily hold more than full permission to a heap location during assertion evaluation, and we show that this semantics provides a formal justification for the rules employed by existing automated verifiers.

3.1. Introduction

As we have seen in the previous chapter, the main connective of separation logic is the *separating conjunction* $*$ (also called the *star*), which permits splitting the resources held by a state: If A and B are SL assertions, the assertion $A * B$ holds in a state σ iff the resources held in σ can be split into two states σ_A and σ_B, written $\sigma = \sigma_A \oplus \sigma_B$, such that A holds in σ_A and B holds in σ_B. Intuitively, $\sigma_A \oplus \sigma_B$ represents the disjoint union of the resources of both states. As an example, the assertion $l_1 \mapsto v_1 * l_2 \mapsto v_2$ describes a state that (separately) owns the two heap locations l_1 (with value v_1) and l_2 (with value v_2).

In all existing variants of SL, states are *bounded*: They cannot own a location l more than once. Consequently, a state σ can be split into $\sigma_A \oplus \sigma_B$ only if σ_A and σ_B own disjoint parts of the heap. Thus, the assertion $l_1 \mapsto v_1 * l_2 \mapsto v_2$ implies that l_1 and l_2 are not aliases. More generally, the assertion $A * B$ implies that A and B describe disjoint parts of the heap. Thanks to the boundedness of states, SL supports important rules such as Frame and Parallel, which we presented in Section 2.5.

The rule Frame enables reasoning locally about a program statement C. If C executes safely in a state that satisfies P and results in a state that satisfies Q, then it will also execute safely in a state that satisfies $P * R$, and it will result in a state that satisfies $Q * R$ (provided that R does not mention variables modified by C). This rule is crucial to prove that

properties of the uninvolved parts of the heap (described by R) are not affected by executing C; they can be *framed around* C. Similarly, the rule Parallel enables reasoning locally about each parallel thread of a parallel composition, given that the two threads operate on disjoint parts of the heap.

To reason about concurrent sharing and the absence of race conditions, SL has been extended with *fractional permissions* [81, 82]. In this setting, a state can own a fraction p of a heap location l, written $l \overset{p}{\mapsto} v$, where p is a positive rational number. Fractional ownership ($p < 1$) grants read access to the location l, while exclusive ownership ($p = 1$) grants read and write access. States are also bounded in this setting, in the sense that they cannot own more than a fraction 1 of a heap location l. Two states σ_A and σ_B can be combined iff their fractional ownerships of each heap location l sum to at most 1 and they agree on the values of the heap locations owned by both. Combined with the rule Parallel, fractional permissions are particularly suitable for reasoning about concurrent threads that read the same heap locations. Consider an example with two concurrent threads. Exclusive ownership of l can be split into half ownership for each thread, which enables both threads to read l, and exclusive ownership of l (and thus write access) can be regained after the two threads have finished executing.

[81]: Boyland (2003), *Checking Interference with Fractional Permissions*
[82]: Bornat et al. (2005), *Permission Accounting in Separation Logic*

3.1.1. Fractional Predicates

SL supports *predicates*[1] more general than the points-to predicate, to enable reasoning about arbitrarily large data structures and at a higher level of abstraction. Ownership of arbitrarily large data structures, such as binary trees or linked lists, can for example be described with *inductively-defined* predicates. Moreover, partial data structures can be expressed with the *separating implication* connective $\mathrel{-\!\!*}$ (also called *magic wand* or *wand*): The predicate $A \mathrel{-\!\!*} B$ describes resources which, combined with any state in which A holds, results in a state in which B holds. It can intuitively typically be seen as expressing the difference in resources between B and A: If B specifies an entire data structure, and A specifies a part of this data structure, then the wand $A \mathrel{-\!\!*} B$ can express ownership of B where A has been removed. Specifying partial data structures with wands has proved useful, for example to track the ongoing iteration over a data structure [173, 174] (where the left-hand side of the wand represents the part of the data structure that remains to be traversed), or to formally reason about borrowing references in the Rust programming language [59]. Magic wands have also been used to abstractly specify protocols on client calls to an API [175–177], such as the protocol that governs Java iterators. We discuss magic wands (and their use cases) in detail in Chapter 4.

Given the importance of these general predicates, it is not surprising that the concept of fractional ownership has been generalized to predicates: If A is an arbitrary SL predicate and π is a fraction, then $\pi \cdot A$ is a *fractional predicate* that represents a fraction π of A. Fractional predicates are supported by automated SL verifiers and have been studied in theory.

Fractional predicates are supported by several automated SL verifiers, including Chalice [152], VerCors [57], VeriFast [15], and Viper [16]. This

1: In this chapter, we use the terms *predicates* and *assertions* interchangeably.

[173]: Maeda et al. (2011), *Extended Alias Type System Using Separating Implication*
[174]: Tuerk (2010), *Local Reasoning about While-Loops*
[59]: Astrauskas et al. (2019), *Leveraging Rust Types for Modular Specification and Verification*
[175]: Krishnaswami (2006), *Reasoning about Iterators with Separation Logic*
[176]: Haack et al. (2009), *Resource Usage Protocols for Iterators.*
[177]: Jensen et al. (2011), *Modular Verification of Linked Lists with Views via Separation Logic.*
[152]: Leino et al. (2009), *A Basis for Verifying Multi-threaded Programs*
[57]: Blom et al. (2017), *The VerCors Tool Set*
[15]: Jacobs et al. (2011), *VeriFast*
[16]: Müller et al. (2016), *Viper*

support relies on the concept of *syntactic multiplication*: A fraction π of $A * B$ is interpreted as a fraction π of A combined with a fraction π of B, i.e. $\pi \cdot (A * B)$ is interpreted as $(\pi \cdot A) * (\pi \cdot B)$. Using this distributivity property, the multiplying fraction can be pushed inside the predicate until it applies to points-to predicates, where $\pi \cdot (l \overset{\alpha}{\mapsto} v)$ is interpreted as $l \overset{\pi \cdot \alpha}{\mapsto} v$.

In theory [83, 84], the fractional predicate $\pi \cdot A$ holds in a state σ iff there exists a state σ_A such that A holds in σ_A and σ corresponds to σ_A where all permission amounts have been multiplied by π, which we write $\sigma = \pi \otimes \sigma_A$. We refer to this definition as the *semantic multiplication*. As an example, if $tree(x)$ represents exclusive ownership of all nodes of a binary tree rooted in x, then $0.5 \cdot tree(x)$ represents half ownership of all nodes of this binary tree.

[83]: Le et al. (2018), *Logical Reasoning for Disjoint Permissions*
[84]: Brotherston et al. (2020), *Reasoning over Permissions Regions in Concurrent Separation Logic*

While the semantic and syntactic multiplications look similar, it turns out that they give two distinct meanings to fractional predicates! Indeed, while the semantic entailment $\pi \cdot (A * B) \models (\pi \cdot A) * (\pi \cdot B)$ holds with both types of multiplication, the dual entailment $(\pi \cdot A) * (\pi \cdot B) \models \pi \cdot (A * B)$, which is direct for the syntactic multiplication, does not hold with the semantic multiplication. The reason is that $\pi \cdot (A * B)$, interpreted with semantic multiplication, might require stronger non-aliasing guarantees than the ones provided by $(\pi \cdot A) * (\pi \cdot B)$, as shown by the following example:

Example 3.1.1 Mismatch between the semantic and the syntactic multiplication**.**
$0.5 \cdot (\texttt{x.f} \mapsto v) * 0.5 \cdot (\texttt{y.f} \mapsto v)$ does not entail $0.5 \cdot (\texttt{x.f} \mapsto v * \texttt{y.f} \mapsto v)$ if interpreted with the semantic multiplication. Indeed, $0.5 \cdot (\texttt{x.f} \mapsto v) * 0.5 \cdot (\texttt{y.f} \mapsto v)$ holds in a state σ with exclusive ownership of `x.f` (with value v) and in which `x` and `y` are aliases. However, $0.5 \cdot (\texttt{x.f} \mapsto v * \texttt{y.f} \mapsto v)$ does not hold in σ, otherwise it would imply (by definition of the semantic multiplication) the existence of a state that exclusively owns `x.f` twice, which is not possible since states are bounded.

Current support for fractional predicates in automated verifiers, based on syntactic multiplication, has never been fully formalized. Worse still, as we show in this chapter, the support in these tools is *not* aligned with the formal models considered in theoretical papers on the same topic. Therefore, it is unclear whether the rules they apply, *e.g.*, to recombine two fractions of a recursively-defined predicate (as we explain next), are sound. In this chapter, we show how to give a fully formal model subsuming the cases supported in practical tools, and going beyond such support to formalize what it can mean to split and recombine more-general predicates, such as magic wands.

3.1.2. Distributivity, Factorizability, and Combinability

[83]: Le et al. (2018), *Logical Reasoning for Disjoint Permissions*
[84]: Brotherston et al. (2020), *Reasoning over Permissions Regions in Concurrent Separation Logic*

As prior work highlights [83, 84], three key properties are needed when reasoning with fractional predicates, which we term *distributivity*, *factorizability*, and *combinability*[2]. We will illustrate shortly on an example why these three properties are necessary. The *distributivity* property holds for

2: Prior work used a different terminology and referred to both the distributivity and factorizability properties as "distributivity" or the "distribution principle".

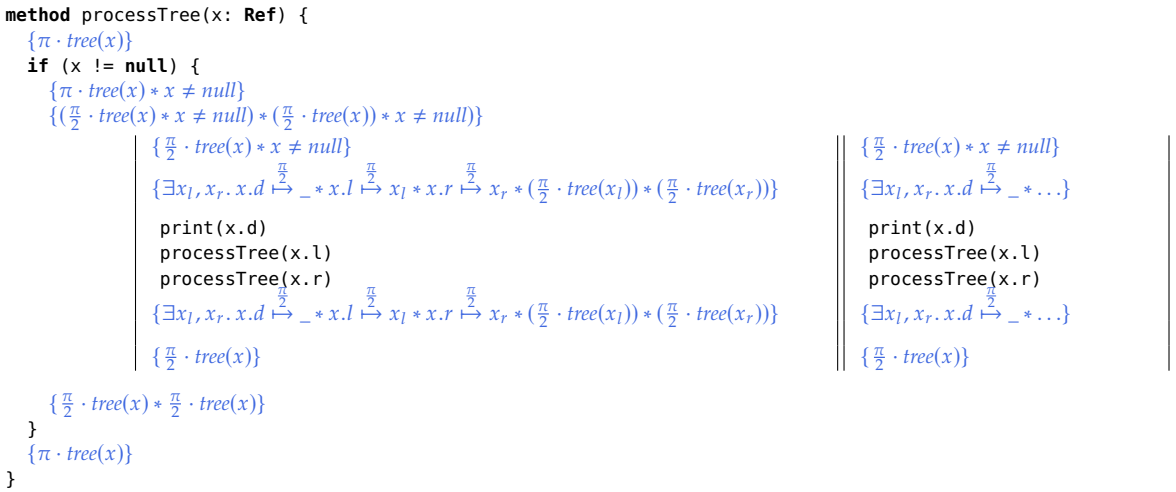

Figure 3.1.: A simple concurrent program that shows why distributivity, factorizability, and combinability are needed when reasoning with fractional resources. The SL predicate $tree(x)$ is recursively-defined as $x \neq null \Rightarrow \exists x_l, x_r.\, x.d \mapsto \_ * x.l \mapsto x_l * x.r \mapsto x_r * tree(x_l) * tree(x_r)$. A proof outline is shown in blue. In SL with semantic multiplication, factorizability does not hold for separating conjunctions and, thus, the entailments at the end of each parallel branch are not valid! With syntactic multiplication, distributivity holds for the separating conjunction by definition, but it has not been shown that combinability holds: it is unclear whether the proof outline is correct. It *is* correct in the semantics we present in this chapter.

a SL connective iff multiplication by any fraction can be distributed over it.[3] Distributivity holds *e.g.*, for the separating conjunction both with semantic multiplication ($\alpha \cdot (A * B)$ entails $(\alpha \cdot A) * (\alpha \cdot B)$) and syntactic multiplication (by definition). However, as we show in Section 3.2, distributivity does *not* hold for the magic wand with semantic multiplication. *Factorizability* is the dual property: it holds for a SL connective iff it is always possible to factor a common fraction out over it. As explained above, factorizability does not hold for the separating conjunction with semantic multiplication, *i.e.*, $(\alpha \cdot A) * (\alpha \cdot B)$ does not always entail $\alpha \cdot (A * B)$. However, factorizability holds for the separating conjunction with syntactic multiplication by definition. Finally, the *combinability* property holds for an assertion A iff two fractions of this assertion can always be combined, *i.e.*, $(\alpha \cdot A) * (\beta \cdot A)$ entails $(\alpha + \beta) \cdot A$. In this case, we say that the predicate A is *combinable*. As we show in Section 3.3, not all predicates are combinable. In particular, even if A and B are combinable, the magic wand $A \mathrel{-\!\!*} B$ is in general not combinable when interpreted with semantic multiplication.

3: Note that, in this chapter, *distributivity* refers to this entailment only, and not to the equivalence, while we refer to the dual entailment as *factorizability*.

To illustrate why these three properties matter when reasoning with fractional resources, consider the simple concurrent program in Figure 3.1, taken from Le and Hobor [83]. This program manipulates the inductively-defined predicate $tree(x) = (x \neq \texttt{null} \Rightarrow \exists x_l, x_r.\, x.d \mapsto \_ * x.l \mapsto x_l * x.r \mapsto x_r * tree(x_l) * tree(x_r))$, which expresses ownership of a binary tree stored on the heap: Either x is `null` (which corresponds to an empty tree), or we have ownership of its fields `x.d` (data of the node), `x.l` (pointer to `x`'s left subtree), and `x.r` (pointer to `x`'s right subtree), and we own the trees rooted in `x.l` and `x.r`. The precondition and postcondition of the method `processTree` is $\pi \cdot tree(x)$ (where π is a ghost parameter omitted from `processTree`'s signature for brevity), which expresses that `processTree` only needs a read access to the tree rooted in `x`, and guarantees the absence of data races. If x is not `null`, `processTree` forks two threads, and both threads print the data of the node (`x.d`), before recursively calling `processTree` on the left and right subtrees of `x`.

[83]: Le et al. (2018), *Logical Reasoning for Disjoint Permissions*

We show in blue a proof outline for this program, which relies on the aforementioned three key properties: distributivity, factorizability, and combinability. This proof outline, explained next, is *invalid* when interpreted with the semantic multiplication, and yet it is successfully verified by tools that implement the syntactic multiplication, such as VeriFast and Viper. If `x` is not `null`, we split $\pi \cdot \mathit{tree}(x)$ into $(\frac{\pi}{2} \cdot \mathit{tree}(x)) * (\frac{\pi}{2} \cdot \mathit{tree}(x))$ to give reading permission to each thread, using the rule Parallel. Inside each thread we unfold the definition of $\mathit{tree}(x)$ and use the **distributivity** property to distribute the fraction π over the separating conjunction, which, conjoined with the knowledge that $x \neq \texttt{null}$, yields $\exists x_l, x_r.\, x.d \overset{\frac{\pi}{2}}{\mapsto} \_ * x.l \overset{\frac{\pi}{2}}{\mapsto} x_l * x.r \overset{\frac{\pi}{2}}{\mapsto} x_r * (\frac{\pi}{2} \cdot \mathit{tree}(x_l)) * (\frac{\pi}{2} \cdot \mathit{tree}(x_r))$. This is enough to justify read access to `x.d` and to recursively call `processTree` on both subtrees (with the ghost parameter $\frac{\pi}{2}$). After the two recursive calls, we use the **factorizability** property to recompose the $\frac{\pi}{2}$ ownership of $\mathit{tree}(x)$ from the $\frac{\pi}{2}$ ownership of its fields and subtrees. Note that as explained above, this step is invalid with the semantic multiplication! Finally, after the two threads have finished executing, we use the **combinability** property to recombine the two $\frac{\pi}{2}$ fractions of $\mathit{tree}(x)$ into $\pi \cdot \mathit{tree}(x)$. Note that to justify this final step, we need to know that $\mathit{tree}(x)$ is combinable. This proof outline illustrates a typical pattern: Distributivity is necessary when we unfold a fractional resource, while factorizability is necessary to fold back the fractional resource, and combinability is necessary to recombine fractions of a resource that was shared between threads.

This example demonstrates the importance of distributivity, factorizability, and combinability, yet traditional separation logics do not fully support them. In SL semantics based on *semantic* multiplication, distributivity does not hold for magic wands, factorizability does not hold for separating conjunctions, and combinability does not hold for magic wands in general (as we show later). Hence, the entailments at the end of the parallel branches in our example are actually *not valid*, as was already pointed out by Le and Hobor [83]. By contrast, tools that implement *syntactic* multiplication happily verify the program, but it has never been shown whether combinability actually holds in this setting and, hence, whether the last entailment is valid.

[83]: Le et al. (2018), *Logical Reasoning for Disjoint Permissions*

3.1.3. State of the Art

Several approaches have been proposed to deal with the limitations of the semantic multiplication.

Factorizability for the separating conjunction. According to Le and Hobor [83], the issue is that predicates such as $\texttt{x.f} \overset{p}{\mapsto} \_ * \texttt{y.f} \overset{p}{\mapsto} \_$, where p is a fractional permission, do not necessarily imply that `x` and `y` are not aliased (for example when $p = 0.5$). They thus use a more complex permission model, the *binary tree share* model [86], which satisfies this *disjointness* property, and define a multiplication over binary tree shares. Going back to Example 3.1.1, if we replace $\frac{1}{2}$ by any binary tree share τ, then we can prove that $\tau \cdot (\texttt{x.f} \mapsto v) * \tau \cdot (\texttt{y.f} \mapsto v)$ entails $\tau \cdot (\texttt{x.f} \mapsto v * \texttt{y.f} \mapsto v)$. More generally, using the disjointness property, they prove that if A and B are τ-*uniform* for some binary tree share

[83]: Le et al. (2018), *Logical Reasoning for Disjoint Permissions*

[86]: Dockins et al. (2009), *A Fresh Look at Separation Algebras and Share Accounting*

τ (meaning that any state that satisfies A or B must have either no permission or exactly τ permission to each and every heap location), then the factorizability entailment $(\pi \cdot A) * (\pi \cdot B) \models \pi \cdot (A * B)$ holds. For example, $\texttt{x.f} \overset{\tau}{\mapsto} v * \texttt{y.f} \overset{\tau'}{\mapsto} v$ is τ-uniform if $\tau' = \tau$ and not otherwise. As well as restricting to τ-uniform predicates, their approach is limited by the complex permission model needed: the notion of multiplication is neither commutative nor left-distributive, and it does not have inverses, which for example prevents factorizability from holding for implications (*i.e.*, $(\pi \cdot P) \Rightarrow (\pi \cdot Q)$ does not necessarily entail $\pi \cdot (P \Rightarrow Q)$).

Brotherston et al. [84] retain fractional permissions, but add new variants of the two main SL connectives. Their assertions include the usual (*weak*) star $*$, the usual (*weak*) wand $-\!\!*$ (adjunct of the weak star), a *strong* star $\circledast$, and (its adjunct) a *strong* wand $-\!\!\circledast$[4]. While the weak star $*$ behaves as usual, the strong star $\circledast$ requires strict non-aliasing, *e.g.*, $\texttt{x.f} \overset{0.5}{\mapsto} v \circledast \texttt{x.f} \overset{0.5}{\mapsto} v$ is unsatisfiable. They then prove valid the factorizability entailment $(\pi \cdot A) \circledast (\pi \cdot B) \models \pi \cdot (A \circledast B)$. To solve the issue in Figure 3.1, they thus redefine the $tree(x)$ predicate with the strong star. They also prove a *strong frame rule* for the strong star, which is quite limited, since it can be applied only with a program statement C that does not receive resources. Moreover, while their strong star satisfies factorizability, their strong wand does not satisfy distributivity.

[84]: Brotherston et al. (2020), *Reasoning over Permissions Regions in Concurrent Separation Logic*

4: Note: the notation here is *opposite* to theirs: They denote the strong star $*$ and the weak star $\circledast$, and analogously for wands.

Combinability. Le and Hobor [83] prove that combinability holds for *precise* predicates [11]. An assertion A is *precise* iff, for any state σ, A holds in *at most one* state σ' smaller than σ. They provide formal rules for proving predicates precise, as well as an induction principle to prove that an inductively-defined predicate is precise, based on a well-founded order of heaps decreasing by at least a constant positive permission amount. As an example, to prove that $tree(x)$ is precise (and, thus, combinable), they can assume that $tree(x_l)$ and $tree(x_r)$ are precise, as long as they can prove that $tree(x_l)$ and $tree(x_r)$ represent heaps smaller than $tree(x)$ by at least a constant positive permission amount, *e.g.*, 1. However, their approach does not capture predicates that are combinable but not precise, which are common in practice, and their induction principle is limited (it cannot be applied for example to inductively-defined predicates with existentially-quantified permission amounts).

[83]: Le et al. (2018), *Logical Reasoning for Disjoint Permissions*

[11]: O'Hearn (2007), *Resources, Concurrency, and Local Reasoning*

Brotherston et al. [84] add *nominal labels* to their assertion language, to track that two fractional predicates have the same *origin*, and thus can be recombined. At any time in a proof, one can conjoin the current assertion A with a fresh label l. Using this label, one can later in the program use the following entailment to recombine two fractions of A: $\alpha \cdot (l \wedge A) * \beta \cdot (l \wedge A) \models (\alpha + \beta) \cdot (l \wedge A)$. They prove the specification $\{l \wedge \pi \cdot tree(x)\}$ `processTree(x)` $\{l \wedge \pi \cdot tree(x)\}$ for some label l for the example in Figure 3.1. To prove such preconditions, they also introduce a *jump modality* @ in their assertion language: intuitively, $@_l A$ means that A holds in the heap labelled by l. While this solves the combinability problem for fractions of an assertion that provably have the same origin, it incurs a significant cost in terms of annotation: their proof outline for the simple method `processTree` (Figure 3.1) requires managing 10 different labels.

[84]: Brotherston et al. (2020), *Reasoning over Permissions Regions in Concurrent Separation Logic*

3.1.4. Approach and Contributions

In this chapter, we present a novel assertion semantics for a separation logic that formally justifies the rules for fractional predicates implemented in automated SL verifiers. Our logic solves the technical problems explained above: Distributivity holds for the magic wand, factorizability holds for the separating conjunction, and the wand $A \mathrel{-\!\!*} B$ preserves combinability (is combinable if B is combinable).

The key idea of our logic is to allow *unbounded* states (states that can have *more than a full permission* to a heap location) in the underlying *assertion* semantics. Bounds on the held permissions are re-introduced in Hoare triples at statement boundaries, which is sufficient to retain SL's powerful reasoning principles, such as the rules Frame and Parallel. In the following, we will refer to our logic as *unbounded separation logic* (*unbounded logic* for short) and to standard SL as the *bounded* logic.

We make the following contributions:

- We present and formalize a novel separation logic that formally justifies the rules for fractional predicates implemented in modern automated SL verifiers. We prove that it guarantees distributivity and factorizability for all commonly-used SL connectives including the star and the magic wand. We show that reimposing boundedness in Hoare triples is sufficient to justify the rules Frame and Parallel (Section 3.2).
- We show that the existing approach of characterizing combinability indirectly via preciseness is limited in general, and prove that commonly-used SL connectives are combinable by defining and reasoning about the property directly. In particular, we prove that, unlike in the bounded logic, the magic wand is combinable[5] in the unbounded logic (Section 3.3).
- We provide a powerful and novel induction principle for reasoning about (co-)inductively-defined predicates in our logic. In particular, this induction principle allows simple justifications that a particular (co-)inductively-defined predicate is combinable (Section 3.4).
- We show how our unbounded logic can serve as a formal foundation to (1) justify and (2) extend the support of fractional resources in automated SL verifiers (such as Chalice, VerCors, VeriFast, and Viper). Using our formal model of syntactic multiplication, we show how to support fractional magic wands, whose support does not exist in any tool, to our knowledge. Moreover, we identify a syntactic criterion on a (potentially recursive) predicate's definition sufficient to ensure that this predicate is combinable (Section 3.5).

5: If its right-hand side is also combinable.

After presenting these technical contributions, we illustrate the advantages of the unbounded logic on two examples of heap-manipulating concurrent programs, one of them from the literature (Section 3.6), and we discuss related work in Section 3.7.

All technical results presented in this chapter have been formalized and proven in Isabelle/HOL [33], and our formalization is publicly available [130, 131].

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

[130]: Dardinier (2022), *Unbounded Separation Logic*

[131]: Dardinier et al. (2022), *Fractional Resources in Unbounded Separation Logic (Artifact)*

3.2. Unbounded Separation Logic

In this section, we present and formally define an *unbounded* version of SL. The key idea, which we explain in Section 3.2.1 and formalize in Section 3.2.2 and Section 3.2.3, is to allow unbounded states (states that can own a heap location more than once) in the assertion semantics. We show in Section 3.2.4 the distributivity and factorizability rules for our unbounded logic. In particular, distributivity holds for the magic wand and factorizability holds for the separating conjunction, which is not the case for traditional, bounded separation logic. Finally, we show in Section 3.2.5 that reimposing boundedness in triples is sufficient to preserve key technical results of CSL, such as the rules Frame and Parallel.

3.2.1. Key Idea: 1+1 = 2

As explained in Section 3.1, one key limitation of reasoning with fractional predicates in (bounded) SL is that factorizing over the star is in general unsound, *i.e.*, the entailment $\pi \cdot A * \pi \cdot B \models \pi \cdot (A * B)$ generally does not hold. As shown with Example 3.1.1 ($\frac{1}{2} \cdot (\mathtt{x.f} \overset{1}{\mapsto} v) * \frac{1}{2} \cdot (\mathtt{y.f} \overset{1}{\mapsto} v) \not\models \frac{1}{2} \cdot (\mathtt{x.f} \overset{1}{\mapsto} v * \mathtt{y.f} \overset{1}{\mapsto} v)$), this entailment fails because of potential aliasing between x and y. The key reason is that states are bounded, and thus the addition of two fractional permissions is a *partial* operation: If a state σ_1 (resp. σ_2) has p_1 (resp. p_2) ownership of the location l, then σ_1 and σ_2 can be combined only if $p_1 + p_2 \leq 1$. In this sense, $1 + 1$ (for example) is undefined. On the left-hand side of this simple example, we add half of a full (1) permission of x.f to half of a full permission of y.f. If x and y are aliases, this corresponds to a permission amount of $\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot 1 = 1$. Since $\frac{1}{2} + \frac{1}{2} \leq 1$, the addition is defined, and thus the left-hand side is satisfiable. On the other hand, the right-hand side is unsatisfiable when x and y are aliases, because the addition is performed *before* the multiplication. In other words, to satisfy the right-hand side when x and y are the same, a state needs $\frac{1}{2} \cdot (1 + 1)$ permission to x.f. But $\frac{1}{2} \cdot (1 + 1)$ is undefined, as $1 + 1$ is undefined as a permission amount.

As explained in Section 3.1.3, Brotherston et al. [84] solve this issue by *strengthening* the left-hand side with the strong star $\circledast$. In other words, they replace $\frac{1}{2} \cdot (\mathtt{x.f} \overset{1}{\mapsto} v) * \frac{1}{2} \cdot (\mathtt{y.f} \overset{1}{\mapsto} v)$ with $\frac{1}{2} \cdot (\mathtt{x.f} \overset{1}{\mapsto} v) \circledast \frac{1}{2} \cdot (\mathtt{y.f} \overset{1}{\mapsto} v)$. This new left-hand side is stronger because, by definition of $\circledast$, it enforces that x and y are non-aliases, and thus implies the right-hand side $\frac{1}{2} \cdot (\mathtt{x.f} \overset{1}{\mapsto} v \circledast \mathtt{y.f} \overset{1}{\mapsto} v)$.

[84]: Brotherston et al. (2020), *Reasoning over Permissions Regions in Concurrent Separation Logic*

In our new unbounded logic, we go the other way, and make the entailment valid by *weakening* the right-hand side. Concretely, we allow $1 + 1$ to equal 2, which makes the right-hand side of Example 3.1.1 satisfiable, and the entailment valid. We achieve this by considering *unbounded* states, *i.e.*, states that can have more than a full permission to a heap location. Going back to the example and proof outline from Figure 3.1, the entailment $\exists x_l, x_r.\, x.d \overset{\frac{\pi}{2}}{\mapsto} \_ * x.l \overset{\frac{\pi}{2}}{\mapsto} x_l * x.r \overset{\frac{\pi}{2}}{\mapsto} x_r * (\frac{\pi}{2} \cdot \mathit{tree}(x_l)) * (\frac{\pi}{2} \cdot \mathit{tree}(x_r)) \models \frac{\pi}{2} \cdot (\exists x_l, x_r.\, x.d \mapsto \_ * x.l \mapsto x_l * x.r \mapsto x_r * \mathit{tree}(x_l) * \mathit{tree}(x_r))$ used in the proof is now valid!

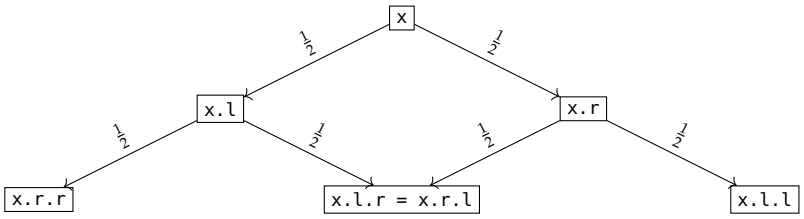

Figure 3.2.: Illustration of the difference between the bounded and unbounded logics. In any assertion semantics that enjoys factorizability, $\frac{1}{2} \cdot \mathit{tree}(x)$ may represent a directed acyclic graph instead of a tree because the upper bound on the permissions held does not prevent sharing (here, of `x.l.r` and `x.r.l`). In unbounded states, this loss of non-aliasing information occurs even for the full $\mathit{tree}(x)$.

Loss of non-aliasing information. Considering unbounded states in the assertion semantics solves the issue of factorizability for the star, but this comes at a cost: As observed by Bornat et al. [82], any assertion semantics that enjoys factorizability weakens the meaning of $\pi \cdot \mathit{tree}(x)$: For example, $\frac{1}{2} \cdot \mathit{tree}(x)$ no longer describes only binary trees, but also admits DAGs (directed acyclic graphs). Figure 3.2 shows an illustration of a state in which $\frac{1}{2} \cdot \mathit{tree}(x)$ holds, even though the central node can be reached from `x` via two distinct paths, `x.l.r` and `x.r.l`.

[82]: Bornat et al. (2005), *Permission Accounting in Separation Logic*

This loss of non-aliasing information caused by factorizability occurs in traditional bounded states (like the one depicted in Figure 3.2) if the sum of the fractional permissions for each heap location does not exceed a full permission. Since unbounded states do not impose an upper bound on permissions, non-aliasing information is lost even for larger fractions: Even the full $\mathit{tree}(x)$ admits DAGs; *e.g.*, consider a variation of Figure 3.2, where each fractional permission is multiplied by 2.

However, a crucial insight of our work is that this loss of non-aliasing information is not an issue in practice. As we will discuss shortly, we re-impose boundedness in a separation logic at *statement* boundaries. That is, even in the unbounded logic, $\mathit{tree}(x)$ denotes a tree before and after each statement. This is sufficient to retain the key rules Frame and Parallel of CSL (as we show in Section 3.2.5), which are by far the most important proof steps that relies on non-aliasing information. As an example, if we split the tree into its left and right subtrees and call a method on the right subtree, we still know that the left subtree will remain unchanged. When the call returns and the subtrees are re-combined, execution is at a statement boundary, and we regain all non-aliasing properties of a tree.

In the rare case that non-aliasing information is needed explicitly (*e.g.*, to prove `x.l.r != x.r.l`), it can be obtained via suitable functional specifications. For instance, the *tree* predicate could be extended to take the set of nodes as an argument and to express that the nodes in the left and in the right subtree are disjoint. We note that many concurrent programs with shared data structures have been formalized and proven correct in the automated SL verifiers Chalice, VerCors, VeriFast, and Viper, even though these verifiers also "suffer" from this loss of non-aliasing information because they use syntactic multiplication and, thus, have factorizability. This empirically supports the claim that explicit non-aliasing information is not crucial for proofs of operations that manipulate data structures to which fractional predicates are held.

3.2.2. State Model and Multiplication

In order to capture different state models and different flavours of SL (such as implicit dynamic frames [70] as seen in Chapter 2), our unbounded logic is parameterized by a *partial commutative monoid* $(\Sigma, \oplus)$ [85, 86]:

[85]: Calcagno et al. (2007), *Local Action and Abstract Separation Logic*
[86]: Dockins et al. (2009), *A Fresh Look at Separation Algebras and Share Accounting*

Definition 3.2.1 Partial commutative monoid**.**
A ***partial commutative monoid*** *(or* ***PCM****) is a pair* $(\Sigma, \oplus)$ *where* Σ *is a set of states and* $\oplus$ *is a partial addition that is commutative and associative.*

For two states $\sigma, \sigma' \in \Sigma$*, we write* $\sigma\#\sigma'$ *to express that* $\sigma \oplus \sigma'$ *is defined, and* $\sigma' \geq \sigma$ *to express that* σ' *is* greater than σ *in the* $\oplus$*-induced order (*i.e.*, iff* $\exists r \in \Sigma.\ \sigma' = \sigma \oplus r$*).*

As an example, we can represent heaps with fractional permissions as partial functions from a set of heap locations L to a set of pairs of a value (from the set V) and a permission amount (from $\mathbb{Q}^+$), *i.e.*, Σ can be the set of functions of type $L \rightharpoonup V \times \mathbb{Q}^+$. Crucially, note that the permission amounts are not upper-bounded. Two states σ and σ' are compatible, *i.e.*, $\sigma\#\sigma'$, iff they agree on the values they both define, and their combination $\sigma \oplus \sigma'$ is the union of their values and the additions of the permission amounts for each heap location. Thus, a state σ' is greater than a state σ iff σ' contains the same value and has at least as much permission as σ for each heap location where σ is defined.

Relation to the IDF algebra

If $(\Sigma, \oplus, |\_|, \mathit{stable}, \mathit{stabilize})$ is an IDF algebra (according to Definition 2.3.1), then $(\Sigma, \oplus)$ is a PCM. However, the two should be instantiated *differently*: The PCM (from Definition 3.2.1) should be instantiated with an *unbounded* state model (*e.g.*, $L \rightharpoonup V \times \mathbb{Q}^+$), whereas the IDF algebra should be instantiated with a *bounded* version of this state model (*e.g.*, $L \rightharpoonup V \times (\mathbb{Q} \cap (0, 1])$).[6]

The unbounded state model is used only to interpret the *syntactic* assertions in the unbounded logic (including inductive and coinductive predicates), as we will see in Section 3.2.3. We then use this interpretation to obtain the *semantic* assertions required by CoreIVL (Section 2.2.1), which are restricted to the bounded states.

6: The instantiation described here is specific to SL with rational permissions. For IDF with real permissions (as presented in Section 2.3.1), we instantiate the unbounded state model for the PCM as $L \rightharpoonup V \times \mathbb{R}_{\geq 0}$, and the bounded state model for the IDF algebra as $\Sigma_{IDF} \triangleq (L \rightharpoonup V \times [0, 1])$.

To express multiplications, our unbounded logic is also parameterized by a *semifield of scalars*:

Definition 3.2.2 Semifield of scalars**.**
A ***semifield of scalars*** *is a tuple* $(S, +, \cdot, 1)$*, which is a semifield with a multiplicative inverse and without a neutral element for the addition. More precisely, for all* $\alpha, \beta, \pi \in S$*, we require* $(S, +, \cdot, 1)$ *to satisfy the following axioms:*

$$\alpha \cdot 1 = \alpha \qquad \alpha \cdot \alpha^{-1} = 1 \qquad \alpha + \beta = \beta + \alpha \qquad \alpha \cdot \beta = \beta \cdot \alpha$$

$$(\alpha \cdot \beta) \cdot \pi = \alpha \cdot (\beta \cdot \pi) \qquad \pi \cdot (\alpha + \beta) = (\pi \cdot \alpha) + (\pi \cdot \beta)$$

Every scalar is required to have a multiplicative inverse, which is crucial

to get some properties, *e.g.*, to factorize a fraction out of an implication. As an example, the set of positive rational numbers $\mathbb{Q}^+$ and set of positive reals $\mathbb{R}^+$ are semifields of scalars. We also require that we can multiply states by scalars with a multiplication operation $\otimes$:

Definition 3.2.3 Left module.
*A **left** S-**module** Σ is a tuple $(\Sigma, \oplus, S, +, \cdot, 1, \otimes)$ where $(\Sigma, \oplus)$ is a PCM, $(S, +, \cdot, 1)$ is a semifield of scalars, and $\otimes$ is a total multiplication from $S \times \Sigma$ to Σ. More precisely, for all $\alpha, \beta \in S$ and $\sigma, \sigma' \in \Sigma$, we require $(\Sigma, \oplus, S, +, \cdot, 1, \otimes)$ to satisfy the the following axioms:*

$$1 \otimes \sigma = \sigma \qquad \alpha \otimes (\beta \otimes \sigma) = (\alpha \cdot \beta) \otimes \sigma$$

$$\alpha \otimes (\sigma \oplus \sigma') = (\alpha \otimes \sigma) \oplus (\alpha \otimes \sigma') \qquad (\alpha + \beta) \otimes \sigma = (\alpha \otimes \sigma) \oplus (\beta \otimes \sigma)$$

In our example, if σ is a partial function from L to $V \times \mathbb{Q}^+$, and π is an element of $\mathbb{Q}^+$, then $\pi \otimes \sigma$ can be defined as multiplying location-wise the permission amounts of σ by π, and leaving the values unchanged.

Finally, we require a predicate *bounded* on Σ, where *bounded*(σ) means that σ is a bounded state. The predicate *bounded* must be (downward) monotonic, *i.e.*, all states smaller than a bounded state must also be bounded. In our example, a state is bounded iff it has at most 1 permission to each heap location.

3.2.3. Assertions

To capture different permission models with our unbounded logic, we consider, in our assertion language, *atomic* semantic assertions (*i.e.*, functions from Σ to Booleans) to abstract over simple SL or IDF assertions that do not contain connectives, such as the usual SL points-to assertion $\texttt{x.f} \overset{p}{\mapsto} v$ or IDF accessibility predicate `acc(x.f, p)`. In this chapter[7], we consider the following syntax for assertions, where A ranges over syntactic assertions, x ranges over variable names, and $\mathcal{B}$ ranges over atomic semantic assertions:

$$A ::= \mathcal{B} \mid A * A \mid A \mathrel{-\!\!*} A \mid \pi \cdot A \mid \varepsilon \cdot A \mid A \Longrightarrow A$$
$$\mid A \wedge A \mid A \vee A \mid \exists x.\, A \mid \forall x.\, A \mid \mathbf{P} \mid \lceil A \rceil$$

The meaning of SL assertions is defined in Figure 3.3. Most connectives are defined in the usual SL[8] $(*, \mathrel{-\!\!*})$ or logical $(\wedge, \vee, \exists, \forall, \Longrightarrow)$ way. The *wildcard* assertion $\varepsilon \cdot A$ represents an unknown (existentially-quantified) fraction of A (recall that scalars are required to have a multiplicative inverse and thus they cannot be zero). Wildcard assertions are ideal to represent read-only duplicable permissions [152]; as an example, $\varepsilon \cdot (\texttt{x.f} \mapsto v)$, represents *some non-zero* permission of `x.f` (which should contain the value v).[9] Note that, to avoid orthogonal issues such as capture-avoidance and clashes between free and bound names, we use a total store, and thus allow the existential and universal quantifiers to "overwrite" values in the store. For example, the assertion $x = 5 \wedge (\exists x.\, x = 7)$ is *satisfiable*, because the existential quantifier "overwrites" the value of the variable x in the store.

7: The syntax presented here is simpler than the one we used to obtain the Viper-Core instantiation in Section 2.3.4. The formal results described in this chapter were adapted to the more complex syntax of Viper assertions in a subsequent student project.

8: Note that the difference in the semantics of the bounded and unbounded logics comes from the state model and not from the assertion semantics itself.

[152]: Leino et al. (2009), *A Basis for Verifying Multi-threaded Programs*

9: In Chapter 2, we used the notation `acc(x.f, _)` to represent such wildcard permissions.

$$
\begin{array}{ll}
\sigma, s, \Delta \models \mathscr{B} & \triangleq \mathscr{B}(\sigma) \\
\sigma, s, \Delta \models A * B & \triangleq (\exists a, b.\, \sigma = a \oplus b \text{ and } a, s, \Delta \models A \text{ and } b, s, \Delta \models B) \\
\sigma, s, \Delta \models A \mathrel{-\!\!*} B & \triangleq (\forall a.\, (a, s, \Delta \models A \text{ and } \sigma \# a) \Longrightarrow \sigma \oplus a, s, \Delta \models B) \\
\sigma, s, \Delta \models \pi \cdot A & \triangleq (\exists a.\, a, s, \Delta \models A \text{ and } \sigma = \pi \otimes a) \\
\sigma, s, \Delta \models \varepsilon \cdot A & \triangleq (\exists a, \pi.\, a, s, \Delta \models A \text{ and } \sigma = \pi \otimes a) \\
\sigma, s, \Delta \models A \Rightarrow B & \triangleq (\sigma, s, \Delta \models A \Longrightarrow \sigma, s, \Delta \models B) \\
\sigma, s, \Delta \models A \wedge B & \triangleq (\sigma, s, \Delta \models A \text{ and } \sigma, s, \Delta \models B) \\
\sigma, s, \Delta \models A \vee B & \triangleq (\sigma, s, \Delta \models A \text{ or } \sigma, s, \Delta \models B) \\
\sigma, s, \Delta \models \exists x.\, A & \triangleq (\exists v.\, \sigma, s(x := v), \Delta \models A) \\
\sigma, s, \Delta \models \forall x.\, A & \triangleq (\forall v.\, \sigma, s(x := v), \Delta \models A) \\
\sigma, s, \Delta \models \mathbf{P} & \triangleq \sigma \in \Delta(s) \\
\sigma, s, \Delta \models \lceil A \rceil & \triangleq (\mathit{bounded}(\sigma) \Longrightarrow \sigma, s, \Delta \models A)
\end{array}
$$

Figure 3.3.: Meaning of unbounded SL assertions. $\sigma \in \Sigma$ is an unbounded state, s is a store of local variables (mapping variable names to values), and Δ is an interpretation (mapping a store to a set of states from Σ).

For simplicity of our formalization, we incorporate recursively-defined predicates (discussed in Section 3.4) via a *single* syntactic predicate symbol **P**. This is not a mathematical limitation (we can encode multiple predicates in a single one with a dedicated argument to "select" the right predicate definition). The symbol **P** represents instances of our (only) predicate; the interpretation context Δ provides the meaning of this predicate: it defines the set of states which correspond to the predicate instance being held. Again for simplicity of our formalization (avoiding a definition for capture-avoiding substitution), parameterization of our predicate symbol is *implicit*: we treat the argument names in a predicate's definition as (reserved) variables in our usual store s, and parameterize Δ with such a store from which it can "read off" the values of (only) these parameters. We then encode an instance of a predicate such as $\mathbf{P}(e)$ via the assertion $\exists x.\, x = e \wedge \mathbf{P}$.

As an example (revisited in Section 3.4) assume that Δ_t represents the predicate *tree*, and the name of the argument of P is x. Then $\Delta_t(s)$ depends only on the value of $s(x)$: $\Delta_t(s)$ represents the set of states that own a tree rooted in $s(\mathsf{x})$. An instance *e.g.*, $\mathit{tree}(x_l)$ is represented as $\exists x.\, x = x_l * \mathbf{P}$. We explain how the interpretation context Δ is constructed in Section 3.4.

Finally, we include a *bounding* operator ($\lceil \_ \rceil$) in our language: The bounded assertion $\lceil A \rceil$ trivially holds in unbounded states, and in all bounded states that satisfy A. This is used to express the usual magic wand in our unbounded logic.

3.2.4. Distributivity and factorizability

We can now prove that all SL connectives satisfy both distributivity and factorizability in our unbounded logic, in contrast to traditional bounded SL. We write $A \models_\Delta B$ to express that A semantically entails B for all possible stores and for the interpretation context Δ, *i.e.*, $(A \models_\Delta B) \triangleq (\forall \sigma, s.\, \sigma, s, \Delta \models A \Rightarrow \sigma, s, \Delta \models B)$. We write $A \equiv_\Delta B$ iff $A \models_\Delta B$ and $B \models_\Delta A$.

We formalize the distributivity and factorizability properties for our assertion language via a set of rules (Figure 3.4). All rules describe equivalences in our logic, except the rules Split and Combine. As explained in Section 3.1, the dual entailment, $(\alpha \cdot A) * (\beta \cdot A) \models_\Delta (\alpha + \beta) \cdot A$, holds for

$$\textsc{DotDot}\quad \beta \cdot (\alpha \cdot A) \equiv_\Delta (\alpha \cdot \beta) \cdot A \qquad \textsc{DotFull}\quad 1 \cdot A \equiv_\Delta A \qquad \textsc{DotStar}\quad \pi \cdot (A * B) \equiv_\Delta (\pi \cdot A) * (\pi \cdot B)$$

$$\textsc{DotWand}\quad \pi \cdot (A \mathrel{-\!\!*} B) \equiv_\Delta (\pi \cdot A) \mathrel{-\!\!*} (\pi \cdot B) \qquad \textsc{DotImp}\quad \pi \cdot (A \Rightarrow B) \equiv_\Delta (\pi \cdot A) \Rightarrow (\pi \cdot B) \qquad \textsc{DotPos}\quad A \models_\Delta B \iff \pi \cdot A \models_\Delta \pi \cdot B$$

$$\textsc{DotExists}\quad \pi \cdot (\exists x.\, A) \equiv_\Delta \exists x.\, (\pi \cdot A) \qquad \textsc{DotForall}\quad \pi \cdot (\forall x.\, A) \equiv_\Delta \forall x.\, (\pi \cdot A) \qquad \textsc{DotAnd}\quad \pi \cdot (A \wedge B) \equiv_\Delta (\pi \cdot A) \wedge (\pi \cdot B)$$

$$\textsc{DotOr}\quad \pi \cdot (A \vee B) \equiv_\Delta (\pi \cdot A) \vee (\pi \cdot B) \qquad \textsc{DotWild}\quad \pi \cdot (\varepsilon \cdot A) \equiv_\Delta \varepsilon \cdot A \equiv_\Delta \varepsilon \cdot (\pi \cdot A) \qquad \textsc{Split}\quad (\alpha + \beta) \cdot A \models_\Delta (\alpha \cdot A) * (\beta \cdot A)$$

$$\textsc{Combine}\quad \frac{\mathit{combinable}_\Delta(A)}{(\alpha \cdot A) * (\beta \cdot A) \models_\Delta (\alpha + \beta) \cdot A} \qquad \textsc{DotPure}\quad \frac{\mathit{pure}(A)}{\pi \cdot A \equiv_\Delta A}$$

Figure 3.4.: Distributivity and factorization rules in the unbounded logic. An assertion A is pure, written $\mathit{pure}(A)$, iff it does not depend on the heap and the interpretation context, *i.e.*, $\forall \sigma, \sigma', s, \Delta, \Delta'.\, (\sigma, s, \Delta \models A \leftrightarrow \sigma', s, \Delta' \models A)$.

combinable assertions only, as shown by the rule Combine, and as we discuss in Section 3.3. We proved the following theorem in Isabelle/HOL:

Theorem 3.2.1 Distributivity and factorizability in the unbounded logic**.**
All rules shown in Figure 3.4 hold in the unbounded logic.

The rules DotImp, and DotPos are notable, since they rely on the key property that the scalars we consider have a multiplicative inverse. In contrast, the tree-permissions from Le and Hobor [83] cannot be inverted, and thus they obtain only one direction for these rules.

[83]: Le et al. (2018), *Logical Reasoning for Disjoint Permissions*

Comparison to bounded SL. The rules DotStar and DotWand do not hold in general in the bounded version of SL. As discussed in Section 3.1, the rule DotStar does not hold because $(\pi \cdot A) * (\pi \cdot B) \models_\Delta \pi \cdot (A * B)$ is not true in general. Similarly, the rule DotWand does not hold, because $\pi \cdot (A \mathrel{-\!\!*} B) \models_\Delta (\pi \cdot A) \mathrel{-\!\!*} (\pi \cdot B)$ is invalid in general. Next, we discuss magic wands in the bounded and unbounded logics in more detail.

A magic wand $A \mathrel{-\!\!*} B$ holds in a state σ iff B holds in all states of the form $\sigma \oplus \sigma_A$, where σ_A is a state compatible with σ and in which A holds. Therefore, one can satisfy a wand in *at least*[10] two ways: (1) by including enough resources in σ such that combining these resources with the ones specified by A results in the resources required by B, or (2) by ensuring that *any* state σ_A in which A holds is incompatible with σ. The latter can be achieved in the bounded logic by including enough resources in σ such that they cannot be combined (in a bounded state) with those already specified by A.

10: As we will see in Chapter 4, there can be more than one way to achieve (1), even when the right-hand side of the magic wand is precise.

Example 3.2.1 Distributivity does not hold for magic wands in the bounded logic**.**
Consider the magic wand

$$W_1 \triangleq \mathtt{x.f} \overset{0.5}{\mapsto} \_ \mathrel{-\!\!*} (\mathtt{x.f} \overset{0.5}{\mapsto} \_ * \mathtt{y.g} \mapsto \_)$$

W_1 holds in a state σ in the bounded logic if (1) σ holds full permission to `y.g`, or (2) σ holds *more* than half (*e.g.*, full) permission to `x.f`. In the latter case, σ combined with any state satisfying the left-hand side of the wand results in a state that holds more than full permission to `x.f`, which is inconsistent in the bounded logic.

To see why distributivity does not hold for the magic wand in the bounded logic, consider the fractional wand $0.5 \cdot W_1$. According to the semantics of fractional predicates (line 3 in Figure 3.3) and strategy (2) above, $0.5 \cdot W_1$ holds in a state σ that holds half permission to `x.f` (and no permission to `y.g`). However, this state does *not* satisfy the wand

$$\begin{aligned} &0.5 \cdot (\texttt{x.f} \overset{0.5}{\mapsto} \_) \mathrel{-\!\!*} 0.5 \cdot (\texttt{x.f} \overset{0.5}{\mapsto} \_ * \texttt{y.g} \mapsto \_) \\ = \; &\texttt{x.f} \overset{0.25}{\mapsto} \_ \mathrel{-\!\!*} (\texttt{x.f} \overset{0.25}{\mapsto} \_ * \texttt{y.g} \overset{0.5}{\mapsto} \_) \end{aligned}$$

because it is not necessarily inconsistent with states σ_A that satisfy its left-hand side and because it does not hold the half permission to `y.g` required by the right-hand side: distributivity does not hold.

In contrast, the unbounded logic offers only strategy (1) to satisfy a wand because states that have more than full permission are no longer necessarily inconsistent. This has three important consequences: First, distributivity (DotWand) holds for W_1 and for wands in general. Second, it makes wands combinable, as we show in Section 3.3. Third, unbounded states lead to a *stronger* meaning for wands compared to the bounded logic as they must be satisfied by following strategy (1).

The stronger meaning of wands in our unbounded logic is not restrictive for many practical purposes. For example the wands used to specify partial data structures during an ongoing traversal or to model borrowing in Rust need to hold according to strategy (1) since proofs use them to obtain the resources on their right-hand sides. Nonetheless, if the bounded version of a wand is really needed it can be expressed in our unbounded logic using our bounding operator: $A \mathrel{-\!\!*} B$ in the bounded logic corresponds to $A \mathrel{-\!\!*} \lceil B \rceil$ in our logic, since $\lceil B \rceil$ trivially holds in unbounded states. Thus, proofs that require the bounded version of magic wands can still be expressed in our logic.

3.2.5. Reimposing Boundedness in CSL Triples

Considering unbounded states in the assertion semantics might at first glance look surprising or even dangerous. After all, non-aliasing is a key component of separation logic, and it is lost with unbounded states: For example, the SL entailment $\texttt{x.f} \mapsto \_ * \texttt{y.f} \mapsto \_ \models_\Delta \texttt{x} \neq \texttt{y}$ does not hold in the unbounded logic.

Nonetheless, we show that our unbounded logic retains non-aliasing reasoning and key technical results such as the rules Frame and Parallel, by adapting the definition of validity of CSL triples (Definition 2.5.1) to only consider bounded states in the definition, and proving in Isabelle that all the CSL rules from Figure 2.10 are still sound with this adapted definition in our unbounded setting.

Definition 3.2.4 Validity of CSL triples in the unbounded logic.
Let $\Sigma \triangleq (L \rightharpoonup V \times \mathbb{R}_{\geq 0})$. *Given a natural number* n, *a ParImp command* C *(as defined in Section 2.5), a store of logical variables* s (i.e., *a partial mapping from variable names in Var to values in Val), an unbounded state* $\omega \in \Sigma$, *and a set of unbounded states with stores* $Q \subseteq \Sigma \times (Var \rightharpoonup Val)$, *we define the predicate* $safe_n(C, s, \omega, Q)$ *exactly as in Definition 2.5.1:*

$safe_0(C, s, \omega, Q)$ *always holds.*

$safe_{n+1}(C, s, \omega, Q)$ *holds iff the following conditions hold:*

1. *If* $C = \textbf{skip}$ *then* $(s, \omega) \in Q$
2. $accesses(C, s) \subseteq readDom(\omega)$ *and* $writes(C, s) \subseteq writeDom(\omega)$
3. *For all heaps* h *and IDF states* ω_f, *if* $\omega \# \omega_f$, $\omega \oplus \omega_f \rightsquigarrow h$, *and* $\mathsf{stable}(\omega_f)$, *then* $\langle C, (s, h)\rangle \not\rightarrow \bot$
4. *For all heaps* h, *IDF states* ω_f, *commands* C', *stores* s', *and heaps* h', *if* $\omega \# \omega_f$, $\omega \oplus \omega_f \rightsquigarrow h$, $\mathsf{stable}(\omega_f)$, *and* $\langle C, (s, h)\rangle \rightarrow \langle C', (s', h')\rangle$, *then there exists another state* ω' *such that* $\omega' \# \omega_f$, $\omega' \oplus \omega_f \rightsquigarrow h'$, $\mathsf{stable}(\omega')$, *and* $safe_n(C', s', \omega', Q)$.

Using this predicate, we define the validity of CSL triples in the unbounded logic as follows:

$$\Delta \models_{CSL} [P]\ C\ [Q]$$
$$\triangleq \big(\forall (s, \omega) \in P.\, \mathsf{stable}(\omega) \wedge bounded(\omega) \Rightarrow (\forall n.\, safe_n(C, s, \omega, Q))\big)$$

We have highlighted in blue the difference with Definition 2.5.1: CSL triples only consider *bounded* initial states ω.[11] Importantly, all states from Σ appearing in this definition (ω, ω', and ω_f) are actually bounded. Indeed, $\omega \oplus \omega_f \rightsquigarrow h$ (resp. $\omega' \oplus \omega_f \rightsquigarrow h'$) implies that $\omega \oplus \omega_f$ (resp. $\omega' \oplus \omega_f$) is bounded (recall that $\omega \rightsquigarrow h$ is defined as $dom(\omega) = dom(h) \wedge (\forall l \in dom(\omega).\, \omega(l) = (h(l), 1))$). Moreover, boundedness is downward-closed, which in turn implies that ω (resp. ω') and ω_f are bounded.

11: Note that our ViperCore instantiation of CoreIVL Section 2.3.4 works slightly differently than what we present here. In our ViperCore instantiation, as explained previously, we first interpret all assertions in the unbounded logic, and then restrict them to bounded states. Definition 2.5.1 with assertions restricted to bounded states results in a definition similar to Definition 3.2.4.

Imaginary separation logic states

Analogously to imaginary numbers, unbounded states and unstable states can be considered *imaginary* SL states. As imaginary numbers were invented to serve as *intermediate* steps towards the *real* solutions of cubic equations of the form $x^3 = ax + b$, unbounded and unstable states serve as *intermediate* steps to give a meaning to assertions that will be used in CSL triples, but the validity of CSL triples considers only *real* SL states (*i.e.*, states that are both stable and bounded).[12] Moreover, as complex numbers give rise to real numbers when their imaginary parts cancel out, unbounded states give rise to bounded states when multiplied by a scalar smaller than 1, and unstable states give rise to stable states when combined with states that own the corresponding permissions.

12: As shown by Definition 3.2.4, SL states (*e.g.*, with fractional permissions) ω can themselves be considered *imaginary* states at a different level, as they give rise to *real* program states (partial heaps) h when combined with another state σ_f such that $\omega \oplus \sigma_f \rightsquigarrow h$.

We have proven in Isabelle versions of Theorem 2.5.2 and Theorem 2.5.1 adapted to our unbounded setting, *i.e.*, we have proven that all rules from Figure 2.10 are sound in this setting, including the key rules Frame and Parallel, and that this definition of validity is adequate. In contrast, the general frame rule does not hold with the strong star $\circledast$ from Brotherston

et al. [84], as it is restricted to program statements C that do not receive any resources (*e.g.*, by acquiring a lock or receiving a message in a concurrent program).

[84]: Brotherston et al. (2020), *Reasoning over Permissions Regions in Concurrent Separation Logic*

We have also proven the following stronger consequence rule, which allows strengthening preconditions and weakening postconditions based on *bounded* entailments:

Proposition 3.2.2 Bounded consequence rule**.**
Assume that

1. $\Delta \models_{CSL} [P]\ C\ [Q]$ *holds,*
2. *for all bounded states* ω, $\omega \in P' \Rightarrow \omega \in P$, *and*
3. *for all bounded states* ω, $\omega \in Q \Rightarrow \omega \in Q'$,

then $\Delta \models_{CSL} [P']\ C\ [Q']$ *holds.*

This *bounded consequence rule* is useful to derive non-aliasing facts. For example, while the entailment $\texttt{x.f} \mapsto \_ * \texttt{y.f} \mapsto \_ \models_\Delta \texttt{x} \neq \texttt{y}$ does not hold in the unbounded setting, all bounded states satisfying $\texttt{x.f} \mapsto \_ * \texttt{y.f} \mapsto \_$ also satisfy $\texttt{x} \neq \texttt{y}$, and thus one can use this rule to "weaken" the postcondition $\texttt{x.f} \mapsto \_ * \texttt{y.f} \mapsto \_$ to the postcondition $\texttt{x.f} \mapsto \_ * \texttt{y.f} \mapsto \_ * \texttt{x} \neq \texttt{y}$.

3.3. Combinable Assertions

As motivated earlier, it is often useful to split some predicate (with the rule Split from Figure 3.4) into two (or more) fractions, and to recombine these fractions later. As illustrated by the example in Figure 3.1, splitting is typically used to enable threads to concurrently read the same heap data structure. Recombining the fractions is then crucial to get back exclusive ownership, and thus to be able to modify the data structure.

However, combining fractions of the same predicate is not always sound, *i.e.*, the entailment $(\alpha \cdot A) * (\beta \cdot A) \models_\Delta (\alpha + \beta) \cdot A$ is in general not valid, as shown by the following example.

Example 3.3.1 A non-combinable assertion**.**
Consider the disjunction $A \triangleq (\texttt{x.f} \mapsto \_ \vee \texttt{x.g} \mapsto \_)$. A holds in a state σ_f (resp. σ_g) with full ownership of `x.f` (resp. `x.g`) and no other ownership. Thus, by definition, $0.5 \cdot A$ holds in $0.5 \otimes \sigma_f$ and $0.5 \otimes \sigma_g$. However, A does *not* hold in the state $(0.5 \otimes \sigma_f) \oplus (0.5 \otimes \sigma_g)$, because this state has only half ownership of both `x.f` and `x.g`, and thus it satisfies neither disjunct of A.

The disjunction A is thus not *combinable*, in the following sense:

Definition 3.3.1 Combinable assertions**.**
An assertion A *is* ***combinable*** *with respect to an interpretation context* Δ, *written* $\text{combinable}_\Delta(A)$, *iff for all scalars* α *and* β, $(\alpha \cdot A) * (\beta \cdot A) \models_\Delta (\alpha + \beta) \cdot A$.

13: We have proven in Isabelle/HOL that the two definitions are equivalent.

$$\frac{\mathit{combinable}_\Delta(A) \quad \mathit{combinable}_\Delta(B)}{\mathit{combinable}_\Delta(A * B)} \qquad \frac{\mathit{combinable}_\Delta(B)}{\mathit{combinable}_\Delta(A \mathrel{-\!\!*} B)} \qquad \frac{\mathit{combinable}_\Delta(A)}{\mathit{combinable}_\Delta(\pi \cdot A)} \qquad \frac{\mathit{combinable}_\Delta(A)}{\mathit{combinable}_\Delta(\varepsilon \cdot A)}$$

$$\frac{\mathit{pure}(A) \quad \mathit{combinable}_\Delta(B)}{\mathit{combinable}_\Delta(A \Rightarrow B)} \qquad \frac{\mathit{combinable}_\Delta(A) \quad \mathit{combinable}_\Delta(B)}{\mathit{combinable}_\Delta(A \wedge B)} \qquad \frac{\mathit{pure}(A) \quad \mathit{combinable}_\Delta(B)}{\mathit{combinable}_\Delta(A \vee B)}$$

$$\frac{\mathit{combinable}_\Delta(A) \quad \mathit{pure}(B)}{\mathit{combinable}_\Delta(A \vee B)} \qquad \frac{\mathit{combinable}_\Delta(A) \quad \mathit{unambiguous}_\Delta(A, x)}{\mathit{combinable}_\Delta(\exists x.\, A)} \qquad \frac{\mathit{combinable}_\Delta(A)}{\mathit{combinable}_\Delta(\forall x.\, A)} \qquad \frac{\mathit{pure}(A)}{\mathit{combinable}_\Delta(A)}$$

Figure 3.5.: Rules for reasoning about combinable (non-recursive) assertions in the unbounded logic.

Informally, if we restrict[13] α and β such that $\alpha + \beta = 1$, an assertion A is combinable iff the set of states that satisfy A is convex, in the sense that for any two states σ and σ' satisfying A, the set of all combinations $(\alpha \otimes \sigma) \oplus ((1 - \alpha) \otimes \sigma')$ (for $0 < \alpha < 1$) all also satisfy A. Intuitively, one can think of the two states σ and σ' as the states satisfying the conjuncts on the left-hand side of combinability, and their combinations as the states satisfying the right-hand side. The set of combinations can be thought of as a line segment between σ and σ'.

As explained in Section 3.1.3, Le and Hobor [83] have proven that *precise* assertions [11] are combinable. Informally, an assertion A is precise iff, for any heap σ, A holds in *at most one* heap smaller than σ. In practice, many useful assertions are combinable but not precise, which shows that checking combinability indirectly via preciseness is too approximate. As a simple example, consider wildcard assertions $\varepsilon \cdot A$, introduced in Section 3.2. Because wildcard assertions are ideal to represent read-only duplicable permissions, they are pervasive in automatic SL verifiers such as VeriFast [15] (see for example Jacobs and Piessens [178]) and Viper [16] (see for example Summers and Müller [158]). Using our definition of combinability, we can simply prove that a wildcard assertion $\varepsilon \cdot A$ is combinable if A is combinable, and this property is effectively assumed by both verifiers. However, wildcard assertions are not precise. Therefore, we focus, in this work, on the combinability property itself instead of using preciseness as a (strictly less useful) proxy. The rules in Figure 3.5 can be used to establish that a (non-recursive) assertion is combinable, as we have proven in Isabelle:

[83]: Le et al. (2018), *Logical Reasoning for Disjoint Permissions*

[11]: O'Hearn (2007), *Resources, Concurrency, and Local Reasoning*

[15]: Jacobs et al. (2011), *VeriFast*

[178]: Jacobs et al. (2011), *Expressive Modular Fine-Grained Concurrency Specification*

[16]: Müller et al. (2016), *Viper*

[158]: Summers et al. (2020), *Automating Deductive Verification for Weak-Memory Programs (Extended Version)*

Theorem 3.3.1 Soundness of the combinability rules.
All rules presented in Figure 3.5 hold in the unbounded logic.

As an example, to prove that $A * B$ is combinable, it suffices to prove that A and B are combinable. The disjunction $A \vee B$ is combinable if one disjunct is pure and the other disjunct is combinable. As shown by Example 3.3.1, $A \vee B$ might not be combinable even if A and B are combinable. The assertion $\exists v.\, A$ is combinable if A is combinable *and* if A is unambiguous in v. Intuitively, this means that, for a given state σ, there is at most one value of v such that A holds in σ, otherwise the existential could act like a (potentially unbounded) disjunction. This rule is crucial to prove that assertions such as $\exists v.\, \mathtt{x.f} \mapsto v * A$ are combinable, provided that A is combinable. Formally, given an interpretation Δ, an assertion A is unambiguous in v, written $\mathit{unambiguous}_\Delta(A, v)$, iff the

following holds:

$$\forall \sigma_1, \sigma_2, s, v_1, v_2.\ \sigma_1 \# \sigma_2 \wedge \sigma_1, s(v := v_1), \Delta \models A \wedge \sigma_2, s(v := v_2), \Delta \models A \Rightarrow v_1 = v_2$$

$\exists v.\, \mathtt{x.f} \mapsto v$ is trivially unambiguous in v. Moreover, if A is unambiguous in v, then $A * B$ is also unambiguous in v. We can thus derive the following useful rule:

$$\frac{\mathit{combinable}_\Delta(A)}{\mathit{combinable}_\Delta(\exists v.\, l \overset{p}{\mapsto} v * A)}$$

The second rule of Figure 3.5 shows a key result: the magic wand is combinable in the unbounded logic, whereas it is not in bounded SL, as shown by the following example:[14]

14: The wand W_1 from Example 3.2.1 is another example of a wand that is not combinable.

Example 3.3.2 Magic wands are not combinable in the bounded logic. Consider the wand

$$W_2 \triangleq \left(\exists v.\, x.f \mapsto v * (v = y \vee v = z) * x.f.g \overset{1/2}{\mapsto} \_\right) \mathrel{-\!\!*} y.g \mapsto \_$$

W_2 can be satisfied in bounded SL by (at least) the two following states:[15]

1. A state σ_y with full permission to `y.g`.
2. A state σ_z with half permission to `y.g` and full permission to `z.g`. Indeed, any state σ_l satisfying the left-hand side of W_2 and compatible with σ_z must have $x.f = y$ (since $x.f = z$ would imply that the combination $\sigma_l \oplus \sigma_z$ has 1.5 permission to `x.f`, which contradicts boundedness), and thus the left-hand provides half permission to `y.g` (via $x.f.g \overset{1/2}{\mapsto} \_$)), while the other half permission to `y.g` comes from σ_z.

15: Any state with non-zero permission to `x.f` would also *trivially* satisfy W_2, because it would be incompatible with any state satisfying the left-hand side of the wand.

Consider now the state $\sigma = 0.5 \otimes \sigma_y \oplus 0.5 \otimes \sigma_z$, which has $\frac{3}{4}$ permission to `y.g` and $\frac{1}{2}$ permission to `z.f`. The state σ does not satisfy W_2, as it does not have enough permission to (1) satisfy the right-hand side directly, or (2) to force $x.f = y$ when combined with the left-hand side.

However, in the unbounded logic, W_2 can be satisfied only by satisfying (1), which ensures that W_2 is combinable.

3.4. Combinable (Co)Inductive Predicates

The previous section provides rules to prove that *non-recursive* assertions are combinable. For example, using the rules from Figure 3.5, it is easy to prove that the assertion $x \neq \mathtt{null} \Rightarrow \exists x_l, x_r.\, x.d \mapsto \_ * x.l \mapsto x_l * x.r \mapsto x_r$ is combinable. However, these rules are not sufficient on their own to prove that (co)inductively-defined predicates are combinable, but this property is required to prove practical examples. For instance, the proof outline in Figure 3.1 is valid (in the unbounded logic) only if $\mathit{tree}(x)$ is combinable. Recall that $\mathit{tree}(x)$ is defined inductively via the following equation: $\mathit{tree}(x) = (x \neq \mathtt{null} \Rightarrow \exists x_l, x_r.\, x.d \mapsto \_ * x.l \mapsto x_l * x.r \mapsto x_r * \mathit{tree}(x_l) * \mathit{tree}(x_r))$. Our goal is to provide the tools and formal foundations to prove by induction that predicates such as $\mathit{tree}(x)$

are combinable: Assuming that $tree(x_l)$ and $tree(x_r)$ are combinable, it would then be straightforward to prove that $tree(x)$ is also combinable, using the recursive definition of $tree(x)$ and the rules from Figure 3.5.

In this section, we formalize the mathematics necessary to enable such intuitive proofs, which turns out to be non-trivial for general SL assertions. In Section 3.4.1, we formalize the meaning of (co)inductive predicates in our assertion language via the concepts of least and greatest fixed points of a recursive equation, and use Knaster-Tarski's theorem to prove that (under monotonicity conditions) these fixed points exist. We also explain why the standard induction principle derived from this theorem is not sufficient to prove that these fixed points are combinable. We then define, in Section 3.4.2, a class of *set-closure properties*, which captures properties such as combinability and *affinity* (an assertion A is *affine* if it is upward-closed)[16]. Moreover, we formalize and prove a novel, simple, and powerful induction principle for set-closure properties and fixed points: If a non-decreasing (defined in Section 3.4.1) function f preserves a set-closure property P, then the least and the greatest fixed point of f satisfy P. This novel induction principle captures the intuition described above. Proving this induction principle requires transfinite induction: We show in Section 3.4.3 why Kleene's fixed point theorem (which does not require transfinite induction) is not sufficient to prove this induction principle for some recursive predicate definitions that can be expressed in our assertion language.

16: Affine assertions are sometimes called *intuitionistic*.

3.4.1. Preliminaries: Monotonic Functions and Existence of Fixed Points

A recursive equation might have zero, some, or infinitely many fixed points. For example, any interpretation for **P** is a fixed point of the recursive equation **P** = **P**, and thus, fixed points of this simple recursive equation are in general not combinable. Two types of fixed points are typically used in SL: The *least fixed point*, and the *greatest fixed point*. Predicates interpreted as a least (resp. greatest) fixed point are referred to as *inductive* (resp. *coinductive*) predicates. Inductive predicates are particularly suitable to describe finite data structures. As an example, the least fixed point of the recursive equation for $tree(x)$ describes all finite binary trees. On the other hand, coinductive predicates can describe infinite data structures, and are useful to describe infinite sets of permissions, for instance, to specify the input/output behavior of reactive programs [179].

[179]: Penninckx et al. (2015), *Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs*

The fixed points we are interested in are *interpretation contexts*, *i.e.*, functions mapping a store of local variables to the set of states satisfying the predicate instance **P** (see Section 3.2.3). As an example, *tree* can be seen as an interpretation context Δ_t, which takes as input a store of local variables (containing in particular a value for the variable x), and outputs the set of states that satisfy $tree(x)$. Moreover, a recursive definition can be described with an assertion, using the symbol **P** for recursive calls. More precisely, given an assertion A (which might contain the symbol **P**) that represents the (potentially recursive) definition of our predicate, we define the interpretation of our predicate as the least or greatest fixed

point of the function f_A, which we define as follows:

$$f_A \triangleq \lambda\Delta.\, \lambda s.\, \{\sigma \mid \sigma, s, \Delta \models A\}$$

The function f_A takes an interpretation context as input Δ and constructs a new interpretation context, by constructing, for any local store s (defining values for the variables corresponding to predicate parameters), the set of states that satisfy A (recall that an interpretation context maps a local store to a set of states), where the meaning of $\mathbf{P}$ is given by the interpretation context Δ. As an example, we can define the interpretation context Δ_t for our *tree* predicate as the least fixed point of f_A, for $A \triangleq (x \neq \texttt{null} \Rightarrow \exists x_l, x_r.\, x.d \mapsto \_ * x.l \mapsto x_l * x.r \mapsto x_r * \mathbf{P}(x_l) * \mathbf{P}(x_r)))$.[17]

17: Recall that $\mathbf{P}$ does not take explicit arguments in our syntax; $\mathbf{P}(x_l)$ (resp. $\mathbf{P}(x_r)$) is syntactic sugar for $\exists x.\, x = x_l * \mathbf{P}$ (resp. $\exists x.\, x = x_r * \mathbf{P}$).

To formally define the meaning of the *least* and *greatest* fixed point of such a function, we need to define an order on interpretation contexts. Informally, an interpretation context Δ is smaller than another interpretation context Δ' iff Δ "semantically entails" Δ'. More precisely:

Definition 3.4.1 Interpretation context**.**
*An **interpretation context** is a function mapping a local store of variables to a set of states from Σ. An interpretation context Δ is **smaller than** another interpretation context Δ', written $\Delta \sqsubseteq \Delta'$, iff $\forall s.\, \Delta(s) \subseteq \Delta'(s)$.*

Lemma 3.4.1 Interpretation contexts form a complete lattice**.**
The set of interpretation contexts equipped with the partial order relation $\sqsubseteq$ is a complete lattice*. In particular, for a family of interpretation contexts S:*

- *The* supremum *(or* join*) of S, written $\sqcup S$, can be obtained as $\sqcup S \triangleq \lambda s.\, \{\sigma \mid \exists \Delta \in S.\, \sigma \in \Delta(s)\}$.*
- *The* infimum *(or* meet*) of S, written $\sqcap S$, can be obtained as $\sqcap S \triangleq \lambda s.\, \{\sigma \mid \forall \Delta \in S.\, \sigma \in \Delta(s)\}$.*

A *fixed point* of a function f is an interpretation Δ such that $f(\Delta) = \Delta$. The *least* (resp. *greatest*) fixed point of a function f is a fixed point that is smaller (resp. larger) than all other fixed points of f, with respect to the partial order $\sqsubseteq$. Knaster-Tarski's theorem states that any *non-decreasing* function f has a least and a greatest fixed point [180].

[180]: Tarski (1955), *A Lattice-Theoretical Fixpoint Theorem and Its Applications.*

Definition 3.4.2 Non-decreasing function**.**
*A function f is **non-decreasing**, written $\text{mono}^+(f)$, iff $\forall \Delta, \Delta'.\, \Delta \sqsubseteq \Delta' \Rightarrow f(\Delta) \sqsubseteq f(\Delta')$.*

We have proven in Isabelle/HOL that the function f_A is non-decreasing if $\mathbf{P}$ occurs only in positive positions in A.

Theorem 3.4.2 Knaster-Tarski fixed point construction**.**
Let $\text{LFP}(f) \triangleq \sqcap\{\Delta \mid f(\Delta) \sqsubseteq \Delta\}$ and $\text{GFP}(f) \triangleq \sqcup\{\Delta \mid \Delta \sqsubseteq f(\Delta)\}$. If $\text{mono}^+(f)$, then $\text{LFP}(f)$ is the least fixed point of f and $\text{GFP}(f)$ is the greatest fixed point of f.

In addition to the existence of a least (and a greatest) fixed point of a function f, this theorem gives us an induction principle for this fixed point: If an interpretation Δ satisfies $f(\Delta) \sqsubseteq \Delta$, then it is greater than

or equal to $LFP(f)$, because $LFP(f)$ is the infimum of the set of such interpretations (a similar induction principle can be derived for $GFP(f)$). This induction principle allows one to prove properties about each individual state of $LFP(f)$, by choosing a relevant Δ. For example, let $\Delta_P(s)$ (for all s) be the set of all states that satisfy a property P (*e.g.*, owning `x.f` with value 5). If f preserves the property P, *i.e.*, $f(\Delta_P) \subseteq \Delta_P$, then $LFP(f) \subseteq \Delta_P$; in other words, all states in $LFP(f)(s)$ (for all s) satisfy P.

However, it does not appear possible to apply this induction principle to the combinability property, since combinability is not a property of *individual states* in a set of states (such as P above), but rather of (unboundedly large) *subsets* of such a set. Combinability concerns the (infinite) space of all combinations of two states (similar to a convexity property, as explained in Section 3.3).

3.4.2. An Induction Principle for (Co)Inductive Predicates and Set-Closure Properties

Given a non-decreasing function f, Theorem 3.4.2 expresses that f has a least fixed point ($LFP(f)$) and a greatest fixed point ($GFP(f)$), and provides induction principles to reason about these fixed points. However, as explained above, these induction principles do not appear sufficient to prove that these fixed points are combinable. On the other hand, Cousot and Cousot [181] have proven that, if $mono^+(f)$, then $LFP(f)$ (resp. $GFP(f)$) can be expressed as the stationary limit of $f^\alpha(\Delta_\bot)$ (resp. $f^\alpha(\Delta_\top)$), where α ranges over ordinals, f^α is defined by transfinite recursion, and $\Delta_\bot$ (resp. $\Delta_\top$) is defined as the *empty* (resp. *full*) interpretation, as given by the following definition.

[181]: Cousot et al. (1979), *Constructive Versions of Tarski's Fixed Point Theorems*

Definition 3.4.3 Empty interpretation, full interpretation, transfinite recursion**.**
*The **empty interpretation**, written $\Delta_\bot$, maps all stores to the empty set $\varnothing$ (representing the assertion false),* i.e., $\Delta_\bot \triangleq (\lambda s.\, \varnothing)$. *The **full interpretation**, written $\Delta_\top$, maps all stores to the universal set Σ (representing the assertion true),* i.e., $\Delta_\top \triangleq (\lambda s.\, \Sigma)$. *Given a function f, f^α (where α is an ordinal) is defined by transfinite recursion as follows:*

- $f^0 \triangleq (\lambda \Delta.\, \Delta)$.
- *For an ordinal α,* $f^{\alpha+1} \triangleq \big(\lambda \Delta.\, f(f^\alpha(\Delta))\big)$.
- *For a limit ordinal γ,* $f^\gamma \triangleq \big(\lambda \Delta.\, \sqcup\{f^\beta(\Delta) \mid \beta < \gamma\}\big)$.

We show in Section 3.4.3 why Kleene's fixed point theorem cannot be applied to prove that a fixed point of some recursive predicate definitions in our assertion language is combinable, which is why we use ordinals and transfinite induction. Using these constructive definitions of $LFP(f)$ and $GFP(f)$, we can express our induction principle for *set-closure properties*, which are defined as follows.

Definition 3.4.4 Set-closure property**.**
A predicate P on interpretation contexts (i.e. a function from interpretation

contexts to Booleans) is a ***set-closure property*** *iff P satisfies the following:*

$$\exists M.\, \forall \Delta.\, (P(\Delta) \iff \forall s.\, (\forall a, b \in \Delta(s).\, M(a,b) \subseteq \Delta(s)))$$

Intuitively, a set-closure property corresponds to being closed under some operation M, which constructs (from two states) a set of states, as shown by the following examples.

Example 3.4.1 Combinability is a set-closure property**.**
An assertion A is combinable iff it is closed under the operation M that informally constructs the line segment between two states, or, more formally, under the operation $M(a,b) \triangleq \{\sigma \mid \exists p, q.\, p + q = 1 \wedge \sigma = p \otimes a \oplus q \otimes b\}$; combinability is thus a set-closure property.

Example 3.4.2 Affinity is a set-closure property**.**
As another example, an assertion A is *affine* iff it is upward-closed (formally corresponding to the operation $M(a,b) \triangleq \{\sigma \mid \sigma \succeq a\}$), which shows that the property of (an assertion) being affine is also a set-closure property.

We have proven the following induction principle for set-closure properties in Isabelle/HOL.

Theorem 3.4.3 Induction principle for set-closure properties**.**
*Let f be a non-increasing function (*i.e.*, mono$^+(f)$) and P a set-closure property. If f preserves P,* i.e.*, $\forall \Delta.\, P(\Delta) \Rightarrow P(f(\Delta))$, then $P(LFP(f))$ and $P(GFP(f))$ hold.*

Proof. Without loss of generality, we show the proof for the least fixed point (the proof for the greatest fixed point is analogous). Let M be such that for all Δ, $P(\Delta) \iff \forall s.\, (\forall a, b \in \Delta(s).\, M(a,b) \subseteq \Delta(s))$.

We proceed by fixpoint induction, *i.e.*, transfinite induction over the iterates $f^\alpha(\Delta_\bot)$ for ordinals α:

Base case. $P(f^0(\Delta_\bot)) \iff P(\Delta_\bot) \iff \forall s.\, (\forall a, b \in \varnothing.\, M(a,b) \subseteq \varnothing)$, which holds trivially.

Successor case. We have to prove that $\forall \Delta.\, P(\Delta) \Rightarrow P(f(\Delta))$, which is one of our assumptions.

Limit case. Let C be a non-empty chain (*i.e.*, a totally-ordered set) such that $\forall \Delta.\, \Delta \in C \Rightarrow P(\Delta)$. We need to prove that $P(\sqcup C)$ holds, *i.e.*, $\forall s.\, (\forall a, b \in (\sqcup C)(s).\, M(a,b) \subseteq (\sqcup C)(s))$. Let s be a store, and $a, b \in (\sqcup C)(s)$. By definition of the supremum, there exist $\Delta_a, \Delta_b \in C$ such that $a \in \Delta_a(s)$ and $b \in \Delta_b(s)$. Since C is a chain, we have either $\Delta_a \sqsubseteq \Delta_b$ or $\Delta_b \sqsubseteq \Delta_a$. Without loss of generality, let us assume that $\Delta_a \sqsubseteq \Delta_b$. In this case, both a and b belong to $\Delta_b(s)$, and since $P(\Delta_b)$ holds, we have $M(a,b) \subseteq \Delta_b(s)$. Since $\Delta_b(s) \subseteq (\sqcup C)(s)$ (by definition of the supremum), we have that $M(a,b) \subseteq (\sqcup C)(s)$, which concludes the cases. □

This theorem justifies the intuitive induction described at the beginning of this section when P is the combinability property: To prove that $tree(x)$ is combinable, we simply have to prove that the assertion $x \neq \texttt{null} \Rightarrow$

$\exists x_l, x_r.\, x.d \mapsto \_ * x.l \mapsto x_l * x.r \mapsto x_r * tree(x_l) * tree(x_r)$ is combinable (corresponding to $P(f(\Delta))$), while assuming that $tree(y)$ is combinable for all y (corresponding to $P(\Delta)$), which we can do using the rules from Figure 3.5. Moreover, this theorem can be easily leveraged in the context of automated SL verifiers, as we show in Section 3.5: If the assertion language for defining predicates recursively is restricted in ways that are standard in such tools, we directly get that all (co)inductive predicates are combinable.

3.4.3. Kleene's Fixed Point Theorem is too Restrictive for SL

Some readers might wonder why we used ordinals and transfinite induction to prove Theorem 3.4.3, instead of (the simpler) Kleene's fixed point theorem. The reason is that Kleene's theorem forces a stronger assumption on f, namely *Scott-continuity*. The theorem states that, if a function f is *Scott-continuous*, then its least fixed point can be computed as the supremum of $f^n(\Delta_\bot)$, where n ranges over natural numbers, and $\Delta_\bot$ is the empty interpretation. Unfortunately, the rich connectives commonly-employed in separation logics easily violate this requirement. Using the universal quantifier or the magic wand in our recursive definition A is enough to make f_A not Scott-continuous. Worse, using the existential quantifier or the separating conjunction in A is enough for f_A to not satisfy the dual property of Scott-continuity, which is required to prove that $GFP(f_A)$ is the infimum of ${f_A}^n(\Delta_\top)$ (where $\Delta_\top$ is the full interpretation). The following example illustrates the problem.

Example 3.4.3 A recursive SL definition that is not Scott-continuous**.** Consider the recursive definition $A \triangleq \varepsilon{\cdot}(\texttt{x.g} \mapsto \_) \mathrel{-\!\!*} (\texttt{x.g} \mapsto \_ \vee 0.5{\cdot}A)$, interpreted in an affine manner.[18] Recall that $\varepsilon \cdot (\texttt{x.g} \mapsto \_)$ represents an unspecified positive permission amount. Let f_A denote the function associated with this recursive definition, and Δ_p an interpretation context such that, for all stores s, a state σ is in $\Delta_p(s)$ iff σ has at least p permission to `x.g`.

f_A is *not* Scott-continuous, and thus Kleene's theorem does not apply. To see why, let us nonetheless compute $f^n(\Delta_\bot)$. Starting from the empty interpretation $\Delta_\bot$, we get $f_A(\Delta_\bot) = \Delta_1$: We need 1 permission of `x.g` to prove the right-hand side of the wand $(\texttt{x.g} \mapsto \_ \vee 0.5 \cdot A) = \texttt{x.g} \mapsto \_$. Then, ${f_A}^2(\Delta_\bot) = f_A(\Delta_1) = \Delta_{0.5}$, since, in this case, we can prove the disjunct $0.5 \cdot A$ to prove the right-hand side. Similarly, ${f_A}^3(\Delta_\bot) = f_A(\Delta_{0.5}) = \Delta_{0.25}$. By induction, we get ${f_A}^{n+1}(\Delta_\bot) = \Delta_{\frac{1}{2^n}}$.

We can now apply Kleene's formula to obtain a *potential* least fixed point: The supremum of $f^n(\Delta_\bot)$ is $\Delta_{>0}$, where a state is in $\Delta_{>0}(s)$ (for all s) iff it has non-zero permission to `x.g`. However, $\Delta_{>0}$ is *not* a fixed point of f_A, since $f_A(\Delta_{>0}) = \Delta_\top$ (the full interpretation). Indeed, in this case, the wand is always trivially satisfied, since the left-hand side implies the right-hand side.

This example shows that Kleene's theorem is too restrictive to justify the existence of a least fixed point for some recursive SL predicate definitions. The situation is similar for Kleene's dual theorem (existence of a greatest

18: By "interpreted in an affine manner", we mean that the assertion A holds in any state that satisfies *at least* the magic wand, *i.e.*, A holds in any state that owns the magic wand and possibly other resources. In *linear* SL (also called *classical*), this interpretation can be obtained by considering $A * true$ instead of A, where the left conjunct captures the wand, and the right conjunct captures the other resources.

fixed point), because of the existential quantifier and the separating conjunction, as shown by the following example:

Example 3.4.4 A recursive SL definition that is not dual Scott-continuous**.**
The greatest fixed point of the recursive equation $A \triangleq \texttt{x.g} \overset{0.5}{\mapsto} \_ * \varepsilon \cdot (\texttt{x.g} \mapsto \_) * 0.5 \cdot A$ is $\Delta_\bot$. However, Kleene's dual formula for the greatest fixed point (*i.e.*, the infimum of $f^n(\Delta_\top)$) yields Δ_1, which is not a fixed point of this equation, because $f_A(\Delta_1) = \Delta_\bot$.

The richer mathematical foundations we provide in this section are needed to enable direct proofs of combinability over general recursively-defined SL predicates.

3.5. Formal Foundations for Fractional Predicates and Magic Wands in Automatic SL Verifiers

Fractional predicates are supported by several automated SL verifiers, such as VerCors [57], VeriFast [15], and Viper [16]. As explained in Section 3.1, this support relies on the concept of *syntactic multiplication*. For example, the semantics of fractional predicates in VeriFast is explicitly defined as follows: "applying a coefficient f to a user-defined predicate is equivalent to multiplying the coefficient of each chunk mentioned in the predicate's body by f" [15]. VerCors and Viper perform a similar syntactic multiplication when *unfolding* (exchanging a predicate instance with its definition, also called *opening*) or *folding* (the reverse operation, also called *closing*) a fractional predicate.

[57]: Blom et al. (2017), *The VerCors Tool Set*
[15]: Jacobs et al. (2011), *VeriFast*
[16]: Müller et al. (2016), *Viper*

[15]: Jacobs et al. (2011), *VeriFast*

However, as shown by Example 3.1.1, there is a mismatch between the syntactic and the semantic multiplication in the bounded logic. Consider $P(x, y) \triangleq (\texttt{x.g} \mapsto \_ * \texttt{y.g} \mapsto \_)$. While $0.5 \cdot P(x, x)$ is equivalent to false in bounded SL if interpreted with the semantic multiplication, the three verifiers allow the user to obtain this fractional predicate instance in exchange for full permission of `x.g`; this behavior is compatible with the semantics of fractional predicates in our novel unbounded logic.

In this section, we show that our unbounded logic can serve as a formal foundation for fractional predicates in automated SL verifiers, since it gives a meaning to the syntactic multiplication performed by these verifiers, and justifies that fractions of the same predicate can be soundly recombined (under some restrictions). Moreover, using the unbounded logic as a formal foundation enables sound extensions of these verifiers, for example to handle fractional magic wands (which, to our knowledge, no verifier supports yet).

In Section 3.5.1, we define a syntactic multiplication over assertions, and show that it is equivalent to the semantic one in the unbounded logic. From this, we derive rules for fractional magic wands, which could easily be automated in VerCors and Viper. We then define, in Section 3.5.2, a simple syntactic restriction on recursive predicate definitions, which ensures the existence of a least and a greatest fixed point. This allows

$$\pi \odot (A * B) \triangleq (\pi \odot A) * (\pi \odot B) \qquad \pi \odot (A \wedge B) \triangleq (\pi \odot A) \wedge (\pi \odot B)$$
$$\pi \odot (A \mathrel{-\!\!*} B) \triangleq (\pi \odot A) \mathrel{-\!\!*} (\pi \odot B) \qquad \pi \odot (A \vee B) \triangleq (\pi \odot A) \vee (\pi \odot B)$$
$$\pi \odot (\alpha \cdot A) \triangleq (\pi * \alpha) \odot A \qquad \pi \odot (\exists x.\, A) \triangleq \exists x.\, (\pi \odot A)$$
$$\pi \odot (\varepsilon \cdot A) \triangleq \varepsilon \cdot A \qquad \pi \odot (\forall x.\, A) \triangleq \forall x.\, (\pi \odot A)$$
$$\pi \odot (A \Rightarrow B) \triangleq (\pi \odot A) \Rightarrow (\pi \odot B) \qquad \pi \odot A \triangleq \pi \cdot A \quad \text{(otherwise)}$$

Figure 3.6.: Definition of the syntactic multiplication over assertions.

$$\textsc{Fold}\ \frac{\mathit{correctRec}_{\top}(A)}{\pi \odot A \models_{FP(A)} \pi \cdot \mathbf{P}} \qquad \textsc{Unfold}\ \frac{\mathit{correctRec}_{\top}(A)}{\pi \cdot \mathbf{P} \models_{FP(A)} \pi \odot A}$$

$$\textsc{CombineFractions}\ \frac{\mathit{comb}(A) \qquad \mathit{correctRec}_{\top}(A)}{\alpha \cdot \mathbf{P} * \beta \cdot \mathbf{P} \models_{FP(A)} (\alpha + \beta) \cdot \mathbf{P}} \qquad \textsc{PackageWand}\ \frac{F * (\pi \odot A) \models_{\Delta} \pi \odot B}{F \models_{\Delta} \pi \cdot (A \mathrel{-\!\!*} B)}$$

$$\textsc{ApplyWand}\ (\pi \odot A) * \pi \cdot (A \mathrel{-\!\!*} B) \models_{\Delta} \pi \odot B$$

Figure 3.7.: Rules for automated SL verifiers. The rules Fold, Unfold, and CombineFractions, justify what existing verifiers do. The rules PackageWand and ApplyWand show how existing verifiers could be extended to support *fractional magic wands*.

us to derive *fold* and *unfold* rules for fractional predicates, based on the syntactic multiplication, which formally justifies what VerCors, VeriFast, and Viper actually do. Finally, we define, in Section 3.5.3, a syntactic restriction on the definition of a predicate, which ensures that this predicate is combinable, using the results from Section 3.3 and Section 3.4.

3.5.1. Syntactic Multiplication and Fractional Magic Wands

Figure 3.6 shows the definition of the syntactic multiplication over assertions, which we write $\pi \odot A$ for a scalar π and an assertion A. The idea of this syntactic multiplication, which corresponds to what the three verifiers do, is straightforward: We push the multiplication inside, until we reach semantic assertions $\mathcal{B}$ or predicate $\mathbf{P}$. The following theorem follows from the distributivity and factorization rules shown in Figure 3.4.

Theorem 3.5.1 Equivalence of the syntactic and semantic multiplication in the unbounded logic**.**
In the unbounded logic, $\pi \cdot A \equiv_{\Delta} \pi \odot A$.

This result justifies the syntactic multiplication performed by the verifiers. Moreover, it can also be leveraged to improve the support for magic wands in automated SL verifiers. Both VerCors and Viper support magic wands [78, 79], via two operations *package* and *apply*, as we will see in Chapter 4. Packaging a wand $A \mathrel{-\!\!*} B$ amounts to exchanging resources that satisfy the wand with an instance of the wand. Applying a wand $A \mathrel{-\!\!*} B$ boils down to giving up an instance of the wand $A \mathrel{-\!\!*} B$ and resources that satisfy A, in exchange for resources that satisfy B. However, neither VerCors nor Viper support packaging and applying fractions of wands.

[78]: Blom et al. (2015), *Witnessing the Elimination of Magic Wands*
[79]: Schwerhoff et al. (2015), *Lightweight Support for Magic Wands in an Automatic Verifier*

$$
\begin{array}{ll}
\mathit{correctRec}_b(\mathscr{B}) \triangleq \top & \mathit{comb}(\mathscr{B}) \triangleq \mathscr{B} \text{ is combinable} \\
\mathit{correctRec}_b(A * B) \triangleq \mathit{correctRec}_(A) \wedge \mathit{correctRec}_b(B) & \mathit{comb}(A * B) \triangleq \mathit{comb}(A) \wedge \mathit{comb}(B) \\
\mathit{correctRec}_b(A \mathrel{-\!\!*} B) \triangleq \mathit{correctRec}_{\neg b}(A) \wedge \mathit{correctRec}_b(B) & \mathit{comb}(A \mathrel{-\!\!*} B) \triangleq \mathit{comb}(B) \\
\mathit{correctRec}_b(\pi \cdot A) \triangleq \mathit{correctRec}_b(A) & \mathit{comb}(\pi \cdot A) \triangleq \mathit{comb}(A) \\
\mathit{correctRec}_b(\varepsilon \cdot A) \triangleq \mathit{correctRec}_b(A) & \mathit{comb}(\varepsilon \cdot A) \triangleq \mathit{comb}(A) \\
\mathit{correctRec}_b(A \Longrightarrow B) \triangleq \mathit{correctRec}_{\neg b}(A) \wedge \mathit{correctRec}_b(B) & \mathit{comb}(A \Longrightarrow B) \triangleq \mathit{pure}(A) \wedge \mathit{comb}(B) \\
\mathit{correctRec}_b(A \wedge B) \triangleq \mathit{correctRec}_b(A) \wedge \mathit{correctRec}_b(B) & \mathit{comb}(A \wedge B) \triangleq \mathit{comb}(A) \wedge \mathit{comb}(B) \\
\mathit{correctRec}_b(A \vee B) \triangleq \mathit{correctRec}_b(A) \wedge \mathit{correctRec}_b(B) & \mathit{comb}(A \vee B) \triangleq (\mathit{comb}(A) \wedge \mathit{pure}(B)) \vee (\mathit{pure}(A) \wedge \mathit{comb}(B)) \\
\mathit{correctRec}_b(\exists x.\, A) \triangleq \mathit{correctRec}_b(A) & \mathit{comb}(\exists x.\, A) \triangleq \mathit{comb}(A) \wedge \mathit{unambiguous}(A, x) \\
\mathit{correctRec}_b(\forall x.\, A) \triangleq \mathit{correctRec}_b(A) & \mathit{comb}(\forall x.\, A) \triangleq \mathit{comb}(A) \\
\mathit{correctRec}_b(\mathbf{P}) \triangleq b & \mathit{comb}(\mathbf{P}) \triangleq \top \\
\mathit{correctRec}_b(\lceil A \rceil) \triangleq \mathit{correctRec}_b(A) & \mathit{comb}(\lceil A \rceil) \triangleq \bot
\end{array}
$$

Figure 3.8.: Syntactic conditions to ensure the existence of a least and greatest fixed point (on the left), and to ensure that an assertion is combinable (on the right). Intuitively, $\mathit{correctRec}_\top(A)$ (resp. $\mathit{correctRec}_\bot(A)$) holds iff recursive calls (*i.e.*, $\mathbf{P}$ in A) appear in positive (resp. negative) positions.

The rules PackageWand and ApplyWand presented in Figure 3.7 show how existing verifiers could be extended to support *fractional magic wands*.

Proposition 3.5.2 Rules for applying and packaging fractional wands**.** *The rules* PackageWand *and* ApplyWand *from Figure 3.7 are sound in the unbounded logic.*

The rule PackageWand states that it is sound to give up the resources specified by F, which satisfy $\pi \odot B$ when combined with $\pi \odot A$, in exchange for a fraction π of the wand $A \mathrel{-\!\!*} B$. On the other hand, the rule ApplyWand states that it is sound to give up a fraction π of a wand $A \mathrel{-\!\!*} B$ and resources that satisfy $\pi \odot A$, in exchange for resources that satisfy $\pi \odot B$. Since these two rules rely on the syntactic multiplication, they could be easily added to VerCors and Viper, which have algorithms to compute F, as we discuss in the next chapter.

3.5.2. Folding and Unfolding Fractions of Recursively-Defined Predicates

To formally justify the way VeriFast, VerCors, and Viper handle recursively-defined predicates, we need to ensure the existence of a fixed point for all predicate definitions accepted by these tools. Indeed, the three verifiers assume that an instance of a recursively-defined predicate is a fixed-point of its recursive definition. All three verifiers enforce recursive calls to appear in positive positions when A is a recursive predicate definition, since (1) none supports implications whose left-hand side are not pure (*i.e.*, specify resources, including predicate instances), and (2) VerCors and Viper[19] do not allow magic wands inside predicate definitions; again, our work presents foundations for extending this support.

We write $\mathit{correctRec}_\top(A)$ iff recursive calls to $\mathbf{P}$ in A happen in positive positions only. The formal definition of $\mathit{correctRec}_\top$ is shown on the left of Figure 3.8. To avoid duplicating rules, we write *FP* to refer indiscriminately to either *LFP* or *GFP*. We have proved in Isabelle/HOL the following lemma:

19: Viper used to allow magic wands inside predicate definitions in both positive and negative positions, but this is now disallowed due to early findings of this work, namely the fact that magic wands are not combinable in the bounded logic.

Lemma 3.5.3 Correct fixed point interpretation.
If $\mathit{correctRec}_{\top}(A)$ *holds, then* $\forall \sigma, s.\, \sigma, s, \mathit{FP}(f_A) \models A \iff \sigma \in \mathit{FP}(f_A)(s)$.

Combining this result with Theorem 3.5.1, we can now prove the following result, which justifies the way VerCors, VeriFast, and Viper handle folding and unfolding fractions of predicates:

Proposition 3.5.4 Rules for folding and unfolding fractional predicates.
The rules Fold *and* Unfold *from Figure 3.7 are sound in the unbounded logic.*

The rule Fold allows one to give up resources that satisfy $\pi \odot A$ in exchange for a fraction π of the predicate instance $\mathbf{P}$, which is defined (co)inductively by the equation $\mathbf{P} = A$. The rule Unfold permits the reverse operation: to exchange a fraction π of the predicate instance $\mathbf{P}$ with resources that satisfy $\pi \odot A$.

3.5.3. Combinability

Finally, we want to leverage results from Section 3.3 and Section 3.4 to prove that the rules used by VerCors, VeriFast, and Viper to combine fractions of predicates are valid in the unbounded logic. Both VerCors and Viper automatically combine fractions of the same predicate instance, which is currently sound (1) because of their restricted assertion languages and (2) because they forbid magic wands inside predicate definitions. Indeed, VerCors and Viper allow disjunctions, existentially-quantified assertions, and negations only of pure assertions. As explained in Section 3.3, the magic wand interpreted in the bounded logic is not combinable in general, and thus allowing wands inside predicate definitions *and* combining fractions of such a predicate instance would be unsound. Note that restriction (2) could be removed by interpreting wands in the unbounded logic.

In contrast, it is possible to write VeriFast predicates that are *not* combinable, *e.g.*, using existential quantifiers. VeriFast thus performs a syntactic analysis on a predicate definition to detect whether this predicate is combinable, and, if it is, VeriFast emits a lemma that permits combining two fractions of this predicate, which is formally justified by our unbounded logic.

To formally justify the behaviors of the three verifiers, we define in Figure 3.8 (on the right) a syntactic condition for an assertion A, $\mathit{comb}(A)$, which ensures that the assertion A is combinable. The predicate *comb* forbids semantic assertions that are not combinable, as well as implications with an impure left-hand side.[20] Moreover, $\mathit{unambiguous}(A, x)$ can be conservatively checked syntactically, using the fact that $\exists v.\, \mathtt{x.f} \mapsto v$ is trivially unambiguous in v, and the entailment $\mathit{unambiguous}(A, x) \implies \mathit{unambiguous}(A * B, x)$ for all A, B, and x. This is, in essence, what VeriFast does.

Finally, note that $\mathit{comb}(\mathbf{P})$ always holds. This way, we can leverage the induction principle from Section 3.4 (Theorem 3.4.3) to prove that predicates (co)inductively-defined with the recursive equation $\mathbf{P} = A$ such that $\mathit{comb}(A)$ holds are combinable. In particular, we have proven in

20: Note that, unlike existing verifiers, we do not forbid disjunctions as long as one of the two disjuncts is pure. Our version is however not more expressive, as any disjunction with a pure disjunct can be rewritten as an implication with a pure left-hand side, by negating the pure disjunct.

Isabelle/HOL the following result, which is used by the three verifiers in some form:

Proposition 3.5.5 Rule for combining fractions of the same predicates. *The rule* CombineFractions *from Figure 3.7 is sound in the unbounded logic.*

3.6. Examples

The example from Figure 3.1 is one illustration of the power and the simplicity of the unbounded logic; with our logic such direct concurrent-separation-logic proofs *just work*, without additional connectives and with essentially no restriction on the assertions involved in the specifications. The entailments inside the two parallel branches are justified by the rules Unfold and Fold, and the last entailment $(\frac{\pi}{2} \cdot tree(x) * \frac{\pi}{2} \cdot tree(x)$ entails $\pi \cdot tree(x))$ is justified by the rule *Combinability* (since the recursive definition of $tree(x)$ satisfies the syntactic condition *comb* defined in Figure 3.8).

In this section, we further illustrate the flexibility and the simplicity of the unbounded logic on two additional examples. The first example motivates the need for factorizability for the magic wand, while the second example, taken from Brotherston et al. [84], shows that the unbounded logic provides an easy and intuitive way to reason about cross-thread data transfer.

[84]: Brotherston et al. (2020), *Reasoning over Permissions Regions in Concurrent Separation Logic*

3.6.1. Concurrently Reading a Subtree and a Tree

Consider the concurrent method `readBoth` in Figure 3.9, which takes as input a reference `x` and an integer `key`. In this simple example, we start with a fraction π of a tree rooted in `x`. We then sequentially look for a subtree of `x` that matches `key`, using the method `find`, where `find` is specified as follows [84, 182]

$$\{\alpha \cdot tree(x)\}\ \texttt{find(x, key)}\ \{\lambda y.\, \alpha \cdot (tree(y) * (tree(y) \mathrel{-\!\!*} tree(x)))\}$$

[84]: Brotherston et al. (2020), *Reasoning over Permissions Regions in Concurrent Separation Logic*
[182]: Cao et al. (2019), *Proof Pearl*

```
method readBoth(x: Ref, key: Int) {
  {π · tree(x)}
  s := find(x, key)
  {π · (tree(s) * (tree(s) -* tree(x)))}
  {π/2 · (tree(s) * (tree(s) -* tree(x))) * π/2 · (tree(s) * (tree(s) -* tree(x)))}
        | {π/2 · (tree(s) * (tree(s) -* tree(x)))}          || {π/2 · (tree(s) * (tree(s) -* tree(x)))}
        | {π/2 · tree(s) * π/2 · (tree(s) -* tree(x))}      || {π/2 · tree(s) * π/2 · (tree(s) -* tree(x))}
        | readTree(s)                                       || readTree(s)
        | {π/2 · tree(s) * π/2 · (tree(s) -* tree(x))}      || {π/2 · tree(s) * π/2 · (tree(s) -* tree(x))}
        | {π/2 · tree(x)}                                   || {π/2 · tree(x)}
        | readTree(x)                                       || readTree(x)
        | {π/2 · tree(x)}                                   || {π/2 · tree(x)}
  {π/2 · tree(x) * π/2 · tree(x)}
  {π · tree(x)}
}
```

Figure 3.9.: A simple concurrent program that looks for a subtree of `x` that matches `key`, and then concurrently reads from both the tree rooted in `x` and in the subtree.

where y is bound to the return variable. $tree(y) \mathrel{-\!\!*} tree(x)$ intuitively expresses the ownership of all nodes of $tree(x)$, except the nodes from its subtree $tree(y)$. Combined with the subtree $tree(y)$, this magic wand gives us a simple way to get back the ownership of the entire tree, $tree(x)$.

Method `readBoth` then forks two threads, and each thread performs some action that first requires read access to the subtree `s`, and then read access to the whole tree `x`. Thus, the method `readTree` requires some fractional ownership of the tree it reads, *i.e.*, it is specified as $\{\alpha \cdot tree(y)\}$ `readTree(y)` $\{\alpha \cdot tree(y)\}$. In this specification, α can be thought of as a ghost parameter; the method can be called for any (non-zero) fractional amount α. Finally, method `readBoth` joins the two threads, and returns the fractional ownership of $tree(x)$ it started with.

Proving that method `readBoth` satisfies its specification is straightforward in our unbounded logic. After the call to `find`, we split the fraction π of $tree(s) * (tree(s) \mathrel{-\!\!*} tree(x))$ into two fractions $\frac{\pi}{2}$, and we give one fraction to each thread, using the rule *Parallel*. In each thread, we then distribute the fraction $\frac{\pi}{2}$ over the star, to justify the call `readTree(s)`.

After this call, we need to justify that we can read the tree rooted in `x`, which we achieve by first *distributing the fraction* $\frac{\pi}{2}$ over the wand $tree(s) \mathrel{-\!\!*} tree(x)$, and then by applying the wand. Crucially, note that this step is invalid in the bounded logic, since the distributivity property does not hold for the wand! Moreover, this step would also be invalid with the weak or the strong wand from Brotherston et al. [84], and even if we used the binary tree share model from Le and Hobor [83]. Finally, since we (syntactically) know that $tree(x)$ is *combinable*, we recombine the two fractions $\frac{\pi}{2}$ of $tree(x)$ after the threads have finished executing, which concludes the proof.

[84]: Brotherston et al. (2020), *Reasoning over Permissions Regions in Concurrent Separation Logic*

[83]: Le et al. (2018), *Logical Reasoning for Disjoint Permissions*

3.6.2. Cross-thread Data Transfer

We also illustrate our unbounded logic on an example from Brotherston et al. [84], which involves message-passing concurrency, with simplified Hoare rules [183–186]: Given a channel c, a message number i, and an associated message invariant R_i^c, the rule to send message i via channel c is $\{R_i^c(x)\}$ `send(c, x)` $\{emp\}$, whereas the rule to receive this message is $\{emp\}$ `y := receive(c)` $\{R_i^c(y)\}$.

[84]: Brotherston et al. (2020), *Reasoning over Permissions Regions in Concurrent Separation Logic*

[183]: Villard et al. (2009), *Proving Copyless Message Passing*

[184]: Bell et al. (2010), *Concurrent Separation Logic for Pipelined Parallelization*

[185]: Leino et al. (2010), *Deadlock-Free Channels and Locks*

[186]: Hobor et al. (2012), *Barriers in Concurrent Separation Logic*

The method `transfer` first creates a binary tree by calling the method `makeTree()`, and then forks two threads. The first thread calls the same method `find` as in the previous example, to find a subtree rooted in `s` that matches the `key`, and sends the reference `s` to the second thread via the channel `ch`. The second thread receives reference `s`, and then modifies the tree rooted in `s` by calling `modify`, which thus requires exclusive ownership of the tree rooted in `s`. After the modification, the second thread notifies the first one, and both terminate. Finally, the tree rooted in `x` is deleted (alternatively, full access could be returned, but this code is from Brotherston et al. [84]).

[84]: Brotherston et al. (2020), *Reasoning over Permissions Regions in Concurrent Separation Logic*

To verify method `transfer`, we need to transmit from the first to the second thread the knowledge that `s` is a node that belongs to the tree rooted in `x`. It is standard to express such information about heap values by adding a second parameter to the predicate *tree*, representing a

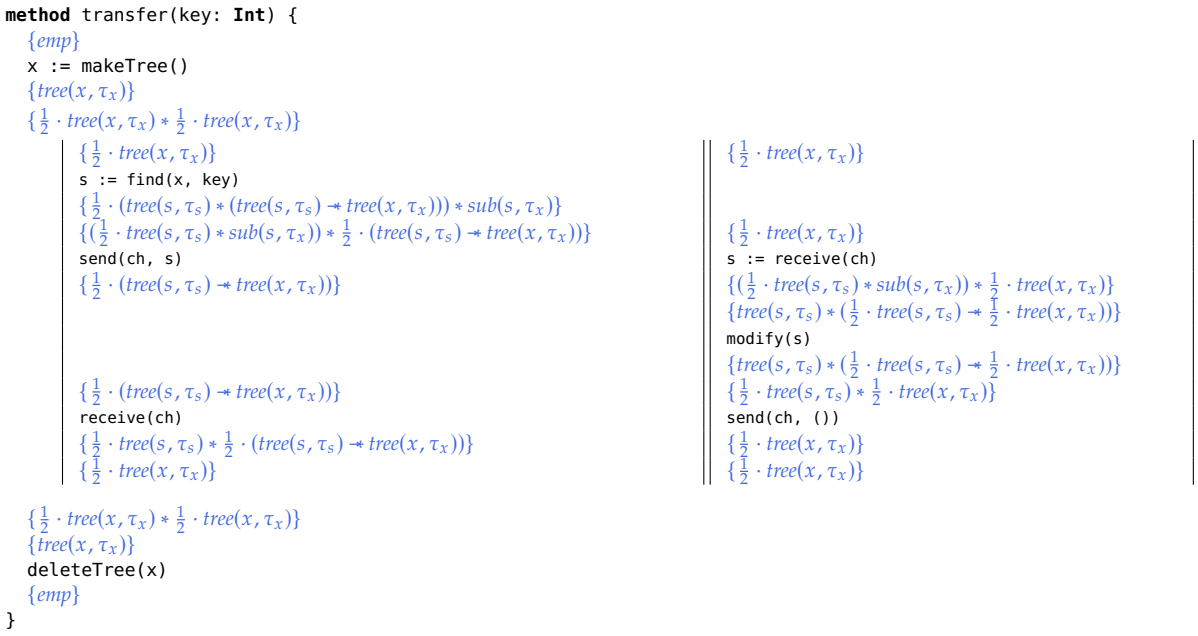

Figure 3.10.: Cross-thread data transfer from Brotherston et al. [84]. `find` is specified as in the previous example. `modify(s)` requires exclusive ownership of the tree rooted in `s`. The first message invariant is $0.5 \cdot tree(s, \tau_s) * sub(s, \tau_x)$, and the second message invariant is $0.5 \cdot tree(s, \tau_s)$.

mathematical abstraction of the tree structure; for a tree rooted at `x`, we will write τ_x for the corresponding mathematical tree. We require our tree abstraction to include the reference identities of each node. The pure function $sub(s, \tau_x)$, which is then easy to write inductively over these mathematical trees, expresses that the reference `s` belongs to the tree τ_x.

The second thread needs this piece of knowledge to prove that $\frac{1}{2} \cdot tree(x, \tau_x)$ can be decomposed into $\frac{1}{2} \cdot tree(s, \tau_s) * (\frac{1}{2} \cdot tree(s, \tau_s) \mathrel{-\!\!*} \frac{1}{2} \cdot tree(x, \tau_x))$ (using a simple inductive lemma), in order to prove it has exclusive ownership of $tree(s, \tau_s)$. Therefore, our first message invariant is $\frac{1}{2} \cdot tree(s, \tau_s) * sub(s, \tau_x)$. After the second thread has received the first message, it can use the aforementioned entailment to justify exclusive ownership of $tree(x, \tau_x)$, and thus call `modify`. After this call, the second thread applies the magic wand to get back half ownership of both $tree(x, \tau_x)$ and $tree(s, \tau_s)$, and it sends $\frac{1}{2} \cdot tree(s, \tau_s)$ via the channel; hence our second message invariant is $\frac{1}{2} \cdot tree(s, \tau_s)$. The first thread then receives $\frac{1}{2} \cdot tree(s, \tau_s)$, and uses the distributivity of the magic wand to get back half ownership of $tree(x, \tau_x)$. Finally, the two threads terminate, and method `transfer` deletes the tree.

Comparison. In contrast to the approach from Brotherston et al. [84], our unbounded logic requires us to add a mathematical tree abstraction to the predicate *tree*, transfer the knowledge that `s` points to a node in τ_x, and prove a simple inductive lemma about tree decomposition. These kinds of reasoning steps are standard in separation logic proofs and required anyway to prove richer functional specifications such as sortedness. Instead, Brotherston et al. use custom *assertion labels* and a *jump modality* in their logic. The first thread transmits the information that the tree rooted in `x` has not been modified since the beginning of the

[84]: Brotherston et al. (2020), *Reasoning over Permissions Regions in Concurrent Separation Logic*

method using the label l_0 introduced in the precondition of `transfer`, via the following invariant for the first message:

$$0.5 \cdot (l_1 \wedge tree(s)) \wedge \left(@_{l_0}^{0.5} 0.5 \cdot ((l_1 \wedge tree(s)) \circledast (l_2 \wand (tree(s) \wand\!\!\circledast\, tree(x))))\right)$$

The left conjunct specifies half of the ownership of the tree rooted in `s`, while the right conjunct contains some knowledge about the initial heap (labelled with l_0).

Our unbounded logic has the advantage that the message invariants are more concise and do not require non-standard connectives in specifications; our first message invariant is $\frac{1}{2} \cdot tree(s, \tau_s) * sub(s, \tau_x)$. This advantage is even better illustrated with the second message invariant, which they specify as

$$0.5 \cdot (l_0 \wedge tree(s)) \wedge l_2 \perp l_3 \wedge \left(@_{l_2}^{0.5} 0.5 \cdot ((l_3 \wedge tree(s)) \wand\!\!\circledast\, (l_4 \wedge tree(x)))\right)$$

where $l_2 \perp l_3$ is another clever but non-standard construct which expresses that the heaps represented by l_2 and l_3 are disjoint. By contrast, our second message invariant is simply $\frac{1}{2} \cdot tree(s, \tau_s)$.

3.7. Related Work

Multiplication with permissions. SL has been extended with different permission models, including fractional permissions [81, 82], counting permissions [82], named permissions [187], and binary tree shares [86]. Although these interoperate well with simple points-to predicates, when considering general fractional predicates, none of them provides the key properties of distributivity, factorizability and combinability. Some of the weaknesses have been previously identified [83, 84], but as we discuss in detail in Section 3.1.3 and Section 3.6.2, the alternatives presented there introduce new complexities to specifications without providing these three properties for their logics in general.

[81]: Boyland (2003), *Checking Interference with Fractional Permissions*
[82]: Bornat et al. (2005), *Permission Accounting in Separation Logic*

[82]: Bornat et al. (2005), *Permission Accounting in Separation Logic*

[187]: Parkinson (2005), *Local Reasoning for Java*

[86]: Dockins et al. (2009), *A Fresh Look at Separation Algebras and Share Accounting*

[83]: Le et al. (2018), *Logical Reasoning for Disjoint Permissions*
[84]: Brotherston et al. (2020), *Reasoning over Permissions Regions in Concurrent Separation Logic*

More-general fractional ownership was (to our knowledge) first explored by Boyland [188], who defines the concept of *nesting*. Nesting enables a heap location l to own some fraction π of a resource A; owning a fraction α of the location l then results in owning a fraction $\alpha \cdot \pi$ of A. Moreover, Boyland permits fractions above 1 in intermediate fractional heaps to get the useful equality $\sigma = \pi \otimes (\frac{1}{\pi} \otimes \sigma)$. However, his work is fundamentally incompatible with SL, because nesting is a static notion in the type system, and because logical and SL connectives such as negations, disjunctions, unrestricted existentials, and magic wands interpreted in the usual way would lead to unsoundness in his framework.

[188]: Boyland (2010), *Semantics of Fractional Permissions with Nesting*

Combinability. As explained in Section 3.1.3, Le and Hobor [83] handle combinability indirectly via preciseness. However, as explained in Section 3.3, preciseness is too restrictive and, for example, does not capture wildcard assertions. Brotherston et al. [84] add labels and jump modalities to the assertion language, which solves the issue of combinability when it can be proven that the two fractions of a resource have the same origin. However, these additional features substantially complicate proofs in the

[83]: Le et al. (2018), *Logical Reasoning for Disjoint Permissions*

[84]: Brotherston et al. (2020), *Reasoning over Permissions Regions in Concurrent Separation Logic*

logic. By contrast, our approach provides simple syntactic rules to prove that an assertion is combinable.

Restricted definitions of magic wands. Boyland [188] defines a connective $-\!\!+$, which is a syntactic connective similar to the magic wand, and which satisfies the analogous combinable property. However, the connective $-\!\!+$ is much more restricted than a magic wand: *e.g.*, $(a = b) -\!\!+ (b = a)$ cannot be proven. Chang and Rival [189] also define a restricted version $A =\!\!* B$ of the magic wand, which is defined inductively and where A and B must be inductive predicates. Intuitively, $A =\!\!* B$ holds in a state σ if one can obtain A via a finite unfolding of predicate instances in B such that resources other than A obtained via the unfolding hold in σ. This restricted wand may satisfy combinability in general (although we have not proved this), but is not as expressive as the general magic wand supported by our work, in particular for expressing arbitrary method contracts.

[188]: Boyland (2010), *Semantics of Fractional Permissions with Nesting*

[189]: Chang et al. (2008), *Relational Inductive Shape Analysis*

Fixed points. Le and Hobor [83] provide an induction principle for heaps with fractional permissions, based on the well-founded order of heaps that decrease by at least a fixed positive permission amount. This induction principle is strictly weaker than the one we present in Section 3.4, since the latter can deal, for example, with recursive predicate instances on the right-hand side of magic wands or wildcard assertions (which may represent arbitrary small permission amounts). Moreover, Section 3.5 shows how to leverage our induction principle to ensure combinability from a simple syntactic condition.

[83]: Le et al. (2018), *Logical Reasoning for Disjoint Permissions*

To give a semantics to abstract predicates, Parkinson and Bierman [190] indirectly construct a semantic predicate environment from an abstract one, by generating a fixed point for a function *step*. As in Section 3.4, it turns out that this *step* function is monotonic but not Scott-continuous, and thus Kleene's fixed-point theorem cannot be applied.

[190]: Parkinson et al. (2005), *Separation Logic and Abstraction*

Step-indexing [191] ensures the monotonicity of recursive definitions by guarding recursive calls with a *later* modality, which is useful for example to deal with recursive types [192]. Step-indexing has been integrated into SL to reason about impredicative protocols [193], and is at the core of Iris [31], a framework for higher-order concurrent SL. It would be interesting to explore how multiplication and the paradigm of unbounded logic presented here can be integrated into a framework such as Iris.

[191]: Appel et al. (2001), *An Indexed Model of Recursive Types for Foundational Proof-Carrying Code*

[192]: Ahmed (2006), *Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types*

[193]: Svendsen et al. (2014), *Impredicative Concurrent Abstract Predicates*

[31]: Jung et al. (2018), *Iris from the Ground Up*

4. Magic Wands

There was a lot more to magic, as Harry quickly found out, than waving your wand and saying a few funny words.

J.K. Rowling,
Harry Potter and the Philosopher's Stone

"We. The scientists. The line of Francis Bacon and the blood of the Enlightenment. Muggles didn't just sit around crying about not having wands, we have our own powers now, with or without magic. If all your powers fail then we will all have lost something very precious, because your magic is the only hint we have as to how the universe must really work - but you won't be left scratching at the ground. Your houses will still be cool in summer and warm in winter, there will still be doctors and medicine. Science can keep you alive if magic fails. It'd be a tragedy, but not literally the end of all the light in the world. Just saying."
Draco had backed up several feet and his face was full of mixed fear and disbelief.
"What in the name of Merlin are you talking about, Potter?"

Eliezer Yudkowsky,
Harry Potter and the Methods of Rationality

The previous chapter introduced a syntax for SL assertions as supported by existing verifiers, along with a novel semantics that justifies the soundness of the rules these verifiers use for fractional predicates.

In this chapter, we focus on the *magic wand* connective $\mathrel{-\!\!*}$ (also known as *separating implication*), and its treatment in existing verifiers. We show that, before our work, all support for magic wands in automated verifiers was either manual or unsound. To address this, we introduce a novel formal foundation for automating magic wands, called *package logic*. This logic captures the broad design space of sound verification algorithms for magic wands, and serves as the basis for our implementation of a sound and automated verification algorithm for magic wands in Viper.

4.1. Introduction

Intuitively, a magic wand $A \mathrel{-\!\!*} B$ can be used to express the difference between the heap locations that B and A provide permission to access. The magic wand is useful, for instance, to specify partial data structures, where B specifies the entire data structure and A specifies a part that is missing [173, 174]. As we saw in the previous chapter (Figure 3.3), the wand $A \mathrel{-\!\!*} B$ holds in a state σ_w, if and only if for *any* program state σ_A in which A holds and that is compatible with σ_w, B holds in the state obtained by combining the heaps of σ_A and σ_w. Thus, if $A * (A \mathrel{-\!\!*} B)$ holds in a state, then so does B, analogously to the *modus ponens* inference rule in propositional logic.

[173]: Maeda et al. (2011), *Extended Alias Type System Using Separating Implication*
[174]: Tuerk (2010), *Local Reasoning about While-Loops*

The magic wand has been shown to enable or greatly simplify proofs in many different cases [59, 173–176, 182, 194, 195]. For instance, Yang [194] uses the magic wand to prove the correctness of the Schorr-Waite graph marking algorithm. Dodds *et al.* [195] employ the wand to specify synchronization barriers for deterministic parallelism. Examples using magic wands to specify partial data structures include tracking ongoing traversals of a data structure [173, 174], where the left-hand side of the wand specifies the part of the data structure yet to be traversed, or for specifying protocols that enforce orderly modification of data structures [175–177] (*e.g.*, the protocol governing Java iterators). More recently, wands have been used for formal reasoning about borrowed references in the Rust programming language, which employs an ownership type system to ensure memory safety [59]. Magic wands concisely represent the *remainder* of a data structure from which a borrowed reference was taken, as well as reflecting back modifications to the part accessible via the reference. Consider for example a struct `Point` (represented by a SL predicate `Point`) with two fields `x` and `y` of type `i32` (represented by the SL predicate `i32`). A Rust method that takes as input a Point `p` and returns a borrow of its field `x` is specified with the postcondition `int32(x) * (int32(x) -* Point(p))`, thus enabling the caller to regain ownership of the entire data structure `Point(p)`.

Given the usefulness of magic wands, it is important for automated SL verifiers to provide automatic support for wands. However, reasoning about a magic wand requires reasoning about *all* states in which the left-hand side holds, which is challenging. It has been shown that a separation logic even without the separating conjunction (but with the magic wand) is as expressive as a variant of second-order logic and, thus, undecidable [196].

Two different approaches [78, 79] that provide partially-automated support are implemented in the verifiers Viper [16] and VerCors [57].[1] However, the approach implemented in VerCors [78] incurs significant annotation overhead, and the approach in Viper [79] suffers from a fundamental, previously undiscovered flaw that renders the approach unsound. Both approaches require hints, in the form of user-provided *package* operations to direct the verifier's proof search. *Packaging* a wand $A \mathrel{-\!\!*} B$ expresses that the verifier should prove and subsequently record $A \mathrel{-\!\!*} B$. To package $A \mathrel{-\!\!*} B$ the verifier must split the current state into two compatible states σ' and σ_w such that $A \mathrel{-\!\!*} B$ holds in σ_w. We call σ_w a *footprint* of the wand. After successfully packaging a wand, the verifier must disallow changes to σ_w to preserve the wand's validity: the verifier *packages the footprint into the wand*.

The key challenge for supporting magic wands in automated verifiers is to define a *package algorithm* that packages a wand. In VerCors's package algorithm [78], a user must manually specify a footprint for the wand and the algorithm checks whether the wand holds in the specified footprint. This leads to a lot of annotation overhead. Viper's current package algorithm [79] reduces this overhead significantly by automatically inferring a suitable footprint. Unfortunately, as we show in this chapter, Viper's current algorithm has a fundamental flaw that causes the algorithm to infer an *incorrect* footprint in certain cases, which may lead to unsound reasoning. We will explain the fundamental flaw

[59]: Astrauskas et al. (2019), *Leveraging Rust Types for Modular Specification and Verification*
[173]: Maeda et al. (2011), *Extended Alias Type System Using Separating Implication*
[174]: Tuerk (2010), *Local Reasoning about While-Loops*
[175]: Krishnaswami (2006), *Reasoning about Iterators with Separation Logic*
[176]: Haack et al. (2009), *Resource Usage Protocols for Iterators.*
[182]: Cao et al. (2019), *Proof Pearl*
[194]: Yang (2001), *An Example of Local Reasoning in BI Pointer Logic: The Schorr-Waite Graph Marking Algorithm*
[195]: Dodds et al. (2011), *Modular Reasoning for Deterministic Parallelism*

[194]: Yang (2001), *An Example of Local Reasoning in BI Pointer Logic: The Schorr-Waite Graph Marking Algorithm*

[195]: Dodds et al. (2011), *Modular Reasoning for Deterministic Parallelism*

[173]: Maeda et al. (2011), *Extended Alias Type System Using Separating Implication*
[174]: Tuerk (2010), *Local Reasoning about While-Loops*

[175]: Krishnaswami (2006), *Reasoning about Iterators with Separation Logic*
[176]: Haack et al. (2009), *Resource Usage Protocols for Iterators.*
[177]: Jensen et al. (2011), *Modular Verification of Linked Lists with Views via Separation Logic.*

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

[196]: Brochenin et al. (2012), *On the Almighty Wand*

[78]: Blom et al. (2015), *Witnessing the Elimination of Magic Wands*
[79]: Schwerhoff et al. (2015), *Lightweight Support for Magic Wands in an Automatic Verifier*

[16]: Müller et al. (2016), *Viper*

[57]: Blom et al. (2017), *The VerCors Tool Set*

1: Based on this chapter, support for magic wand was recently added to the Gillian verifier [64, 197].

[78]: Blom et al. (2015), *Witnessing the Elimination of Magic Wands*

[79]: Schwerhoff et al. (2015), *Lightweight Support for Magic Wands in an Automatic Verifier*

[78]: Blom et al. (2015), *Witnessing the Elimination of Magic Wands*

[79]: Schwerhoff et al. (2015), *Lightweight Support for Magic Wands in an Automatic Verifier*

in Section 4.2; it illustrates the subtlety of supporting this important connective.

Approach and Contributions

In this chapter, we present a formal foundation for sound package algorithms, and we implement a novel such algorithm based on these foundations. Our algorithm requires the same annotation overhead as the prior, flawed Viper algorithm, which is (to our knowledge) the most automatic existing approach. We introduce a formal framework expressed via a novel *package logic* that defines the design space for package algorithms. The soundness of a package algorithm can be justified by showing that the algorithm finds a proof in our package logic. The design space for package algorithms is large since there are various aspects that affect how one expresses the algorithm including (1) which footprint an algorithm infers or checks (there are often multiple options, as we show in Section 4.3), (2) the state model (which differs between different SL verifiers), and (3) restricted definitions of wands (for instance, to ensure each wand has a unique minimal footprint). Our package logic deals with (1) by capturing all sound derivations for the same wand. To deal with (2) and (3), our logic is parametric along multiple dimensions. For instance, we only assume the state model to be an IDF algebra (Definition 2.3.1).

Our logic also supports parameters to restrict the allowed footprints for wands in systematic ways. Such restrictions are useful, for instance, to ensure that magic wands are combinable (as defined in Section 3.3). In this chapter, we introduce a novel restriction of magic wands that ensures they are combinable,[2] and we develop a corresponding second package algorithm, again based on the formal framework provided by our package logic.

2: The definition of combinable wands we present in this chapter is different from the definition from the previous chapter based on the unbounded logic. We will discuss the differences between the two definitions in Section 4.4.

We make the following contributions:

- We formalize a *package logic* that can be used as a basis for a wide range of package algorithms (Section 4.3). The logic has multiple parameters including: an IDF algebra (Definition 2.3.1) to model the states and a parameter to restrict the definition of a wand in a systematic way. We formally prove the logic sound and complete for any instantiation of the parameters in Isabelle/HOL. [132]
- We develop a novel, restricted definition of a wand (Section 4.4), prove in Isabelle/HOL that this wand is combinable [133][3], and show how the generalized version of our package logic can be used to package combinable wands.
- We implement sound package algorithms for both the standard and the combinable wand in the Viper verifier and justify their soundness directly via our package logic (Section 4.5). We evaluate both algorithms on the Viper test suite. Our evaluation shows that (1) our algorithms perform similarly well to prior work and correctly reject examples where prior work is unsound, and (2) our combinable wand definition is expressive enough for most examples.

[132]: Dardinier (2022), *Formalization of a Framework for the Sound Automation of Magic Wands*

[133]: Dardinier (2022), *A Restricted Definition of the Magic Wand to Soundly Combine Fractions of a Wand*

3: If the right-hand side of the wand is combinable.

Our Isabelle formalization and the implementation of our new package algorithm are publicly available [132–134].

[132]: Dardinier (2022), *Formalization of a Framework for the Sound Automation of Magic Wands*

[133]: Dardinier (2022), *A Restricted Definition of the Magic Wand to Soundly Combine Fractions of a Wand*

[134]: Dardinier et al. (2022), *Sound Automation of Magic Wands (Artifact)*

```
1  method leftLeaf(x: Ref) returns (y: Ref)
2    requires tree(x)
3    ensures tree(x)
4  {
5    y := x
6    package tree(x) −∗ tree(x)
7    while (y.left != null)
8      invariant tree(y) ∗ (tree(y) −∗ tree(x))
9    {
10     y := y.left
11     package tree(y) −∗ tree(x)
12       // { hints for package}
13   }
14   apply tree(y) −∗ tree(x)
15 }
```

Figure 4.1.: A simple example of a tree traversal using wands. The code on the left finds the leftmost leaf of a binary tree and includes specifications to prove memory safety. The predicate $tree(x)$ is defined inductively as $\mathbf{acc}(x.val) * \mathbf{acc}(x.left) * \mathbf{acc}(x.right) * (x.left \neq null \Rightarrow tree(x.left)) * (x.right \neq null \Rightarrow tree(x.right))$.The loop invariant uses a wand to summarize the permissions of the input tree excluding the tree not yet traversed. The blue operations are ghost operations to guide the verifier; we omit those specific to predicates. The `package` requires further hints in existing approaches, which we present in the extended version of this chapter [198].

4.2. Motivation

4.2.1. A Typical Example Using Magic Wands

Figure 4.1 shows a simplified version of an example from the VerifyThis competition [199]. The method `leftLeaf` iteratively computes the leftmost leaf of a binary tree (the `package` and `apply` operations, shown in blue, will be explained in Section 4.2.2). The pre- and postconditions of `leftLeaf` are both $tree(x)$, where $tree(x)$ is defined recursively as $tree(x) \triangleq (\mathbf{acc}(x.val) * \mathbf{acc}(x.left) * \mathbf{acc}(x.right) * (x.left \neq null \Rightarrow tree(x.left)) * (x.right \neq null \Rightarrow tree(x.right)))$.[4] Proving this specification amounts to proving that `leftLeaf` is memory-safe and that the permissions to the input tree are preserved, enabling further calls on the same tree, for example to modify or delete the tree afterwards.

[199]: Huisman et al. (2015), *VerifyThis 2012*

4: Note that, apart from being written in IDF and not SL, this definition is slightly different from the one in Chapter 3, as the definition here ensures that the tree is not empty.

The key challenge when verifying `leftLeaf` is specifying an appropriate loop invariant. The loop invariant must track the permissions to the subtree rooted at `y` that still needs to be traversed, since otherwise dereferencing `y.left` in the loop body is not allowed. Additionally, the invariant must track all of the remaining permissions in the input tree rooted at `x` (the permissions to the nodes already traversed and others unreachable from `y`), since otherwise the postcondition cannot be satisfied. The former can be easily expressed with $tree(y)$. The latter can be elegantly achieved with a magic wand $tree(y) \mathrel{-\!\!*} tree(x)$. This wand promises $tree(x)$ if one combines the wand with $tree(y)$. That is, the wand represents (at least) the difference between the permissions making up the two trees. Using SL's modus-ponens-like inference rule ApplyWand (directed by the `apply` operation on line 14, explained next), one can show that the loop invariant entails the postcondition.

4.2.2. Wand Ghost Operations

Automated SL verifiers such as Gillian [30], VeriFast [15], VerCors, and Viper generally represent permissions owned by a program state in two ways: by recording predicate instances (such as $tree(x)$ in Figure 4.1) and *direct* permissions to heap locations. Magic wand instances provide a third way to represent permissions and are recorded analogously. Verifiers that support them require two wand-specific *ghost operations*,

[30]: Fragoso Santos et al. (2020), *Gillian, Part i*

[15]: Jacobs et al. (2011), *VeriFast*

which instruct the verifiers when to prove a wand and when to apply a recorded wand instance using SL's modus-ponens-like rule.

A `package` ghost operation, which corresponds to the rule PackageWand from Section 3.5.1 with $\pi = 1$, expresses that a verifier should prove a new wand instance in the current state and report an error if the proof attempt fails. To prove a new wand instance, the verifier must split the current state into two states σ' and σ_w such that the wand holds in the *footprint* state σ_w; on success, permissions in the footprint are effectively exchanged for the resulting magic wand instance. We call a procedure that selects a footprint by splitting the current state a *package algorithm*. On lines 6 and 11 of Figure 4.1, new wands are packaged to establish and preserve the invariant, respectively.

The `apply` ghost operation, which corresponds to the modus-ponens-like rule ApplyWand from Section 3.5.1 with $\pi = 1$, *applies* a wand $A \mathrel{-\!\!*} B$ if the verifier records a wand instance of $A \mathrel{-\!\!*} B$ and A holds in the current state (and otherwise fails), exchanging these for the assertion B. The `apply` operation is directly justified by the wand's semantics: Combining a wand's footprint with *any* state in which A holds is guaranteed to yield a state in which B holds. For the `apply` operation on line 14 of Figure 4.1, the verifier removes the applied wand instance and $\mathit{tree}(y)$, in exchange for the predicate instance $\mathit{tree}(x)$.

4.2.3. The Footprint Inference Attempt (FIA)

Package algorithms differ in how a footprint for the specified magic wand is selected. In VerCors [57], the user must manually provide the footprint and the algorithm checks whether the specified footprint is correct. In Viper's current approach [79], a footprint is inferred. We explain and compare to the latter approach since it is the more automatic of the two; hereafter, we refer to its package algorithm as *the Footprint Inference Attempt* (*FIA*). Inferring a correct footprint is challenging due to the complexity of the wand connective. In particular, we have discovered that, in certain cases, the FIA infers *incorrect* footprints, leading to unsound reasoning.[5] The goal of this subsection is to understand the FIA's key ideas, which our solution will build on, and why it is unsound.

[57]: Blom et al. (2017), *The VerCors Tool Set*

[79]: Schwerhoff et al. (2015), *Lightweight Support for Magic Wands in an Automatic Verifier*

5: This unsoundness might not be observable in restricted logics, but it is in Viper (as we show in Appendix A.2) and the rich logics supported by existing verification tools.

In general, there may be multiple valid footprints for a magic wand $A \mathrel{-\!\!*} B$. The FIA attempts to infer a footprint which is as close as possible to the *difference* between the permissions required by B and A, taking as few permissions as possible while aiming for a footprint compatible with A (so that the resulting wand can be later applied) [79]. That is, the FIA includes only permissions in the footprint it infers that are specified by B *and not* guaranteed by A.

[79]: Schwerhoff et al. (2015), *Lightweight Support for Magic Wands in an Automatic Verifier*

For a wand $A \mathrel{-\!\!*} B$, the FIA constructs an arbitrary state σ_A that satisfies A (representing σ_A symbolically). Then, the FIA tries to construct a state σ_B in which B holds by taking permissions (and copying corresponding heap values) from σ_A if possible and the current state otherwise. If this algorithm succeeds, the (implicit) inferred footprint consists of the permissions that were taken from the current state. The FIA constructs σ_B by iterating over the permissions and logical constraints in B. For each permission, the FIA checks whether σ_A owns the permission. If so, the FIA adds the permission to σ_B and removes the permission from σ_A.

Otherwise, the FIA removes the permission from the current state or fails if the current state does not have the permission. For each logical constraint, the FIA checks that the constraint holds in σ_B as constructed so far.

Unsoundness of the FIA. We have discovered that for some wands $A \mathrel{-\!\!*} B$, the FIA determines an *incorrect* footprint for the magic wand. This unsoundness can arise when the FIA performs a case split on the content of the arbitrary state σ_A satisfying A. In such situations, the FIA infers a footprint for each case *separately*, making use of properties that hold in that case. For certain wands, this leads to different footprints being selected for each case, while *none* of the inferred footprints can be used to justify B in *all* cases, *i.e.*, for *all* states σ_A that satisfy A. As a result, the packaged wand does *not* hold in any of the inferred footprints, which can make verification unsound, as we illustrate below.

Example 4.2.1 Example showing the unsoundness of the FIA.
The wand

$$w \triangleq \mathbf{acc}(x.f) * (x.f = y \vee x.f = z) \mathrel{-\!\!*} \mathbf{acc}(x.f) * \mathbf{acc}(x.f.g)$$

illustrates the problem. For this wand, every state σ_A satisfying the left-hand side must have permission to `x.f`, and `x.f` must either point to `y` or `z`. If `x.f` points to `y` in σ_A, then to justify the right-hand side's second conjunct, the footprint must contain permission to `y.g`. Analogously, if `x.f` points to `z` in σ_A, then the footprint must contain permission to `z.g`. The wand's semantics requires a footprint to justify the wand's right-hand side for all states in which the left-hand side holds, and thus, a correct footprint must be able to justify *both* cases. Hence, the footprint must have permission to *both* `y.g` and `z.g`. However, the FIA's inferred footprint is in effect the disjunction of these two permissions.

Packaging the above wand w using the FIA leads to unsound reasoning. After the incorrect package described above in a state with permission to `x.f`, `y.g`, and `z.g`, the assertion $\mathbf{acc}(x.f) * (\mathbf{acc}(y.g) \vee \mathbf{acc}(z.g)) * w$ can be proved since the FIA removes permission to either `y.g` or `z.g` from the current state, but not both. However, this assertion does not actually hold! According to the semantics of wands, w's footprint must include permission to `x.f` or permission to both `y.g` and `z.g`, which implies that the assertion $\mathbf{acc}(x.f) * (\mathbf{acc}(y.g) \vee \mathbf{acc}(z.g)) * w$ is equivalent to false.

The unsoundness of the FIA shows the subtlety and challenge of developing sound package algorithms. Algorithms that soundly infer a single footprint for all states in which the wand's left-hand side holds must be more involved than the FIA. Ensuring their soundness requires a *formal* framework to construct them and justify their correctness. We introduce such a framework in the next section.

4.3. Specialized Package Logic

In this section and the next, we present a new logical framework that defines the design space for (sound) package algorithms. The core of this framework is our *package logic*, which defines the space of potential algorithmic choices of a footprint for a particular magic wand. Successfully packaging a wand in a given state is (as we will show) equivalent to finding a derivation in our package logic, and any sound package algorithm must correspond to a proof search in our logic. In particular, we provide soundness (Theorem 4.3.1) and completeness (Theorem 4.3.2) results for our logic. We define a specific package algorithm with this logic at its foundation, inspired by the FIA package algorithm [79] (described in Section 4.2.3) but amending its unsoundness, resulting in (to the best of our knowledge) the first sound and automated package algorithm. This section focuses on the package logic specialized for the standard semantics of magic wands. The general version of the package logic will be presented in Section 4.4.

[79]: Schwerhoff et al. (2015), *Lightweight Support for Magic Wands in an Automatic Verifier*

All definitions and results in this section and the next have been fully mechanized [132] in Isabelle/HOL. Our mechanized definitions are parametric with the underlying verification logic in various senses: the underlying IDF algebra is a parameter, the syntax of assertions is defined in a way which allows simple extension with different base cases and connectives, and the semantics of magic wands itself can be restricted if only particular kinds of footprint are desired in practice (as we will see in Section 4.4).

[132]: Dardinier (2022), *Formalization of a Framework for the Sound Automation of Magic Wands*

4.3.1. Footprint Selection Strategies

As we explained in Section 4.1, there is a wide design space for package algorithms; in particular, many potential strategies for finding a magic wand's footprint exist and none is clearly optimal. Recall that a footprint is a state, and thus consists of permissions to certain heap locations as well as storing their corresponding values; for simplicity we identify a footprint by the permissions it contains.

Example 4.3.1 A magic wand with incomparable footprints**.**
Consider the following magic wand

$$\mathbf{acc}(x.b, 1/2) \mathrel{-\!\!*} \mathbf{acc}(x.b, 1/2) * (x.b \Rightarrow \mathbf{acc}(x.f))$$

Suppose this magic wand is to be packaged in a state where full permissions to both `x.b` and `x.f` are held, and the value of `x.b` is currently false. Two valid potential footprints are:

1. Full permission to `x.f`. This is sufficient to guarantee the right-hand side will hold regardless of the value that `x.b` has by the time the wand is applied.
2. Half permission to `x.b`. By including this permission, the fact that `x.b` is currently false is also included, and thus permission to `x.f` is not needed.

There is no clear reason to prefer one choice over the other: different package algorithms (or manual choices) might choose either. Our package

logic allows either choice along with any of many less optimal choices, such as taking both permissions. On the other hand, as motivated earlier in Section 4.3.1, our package logic must (and does) enforce that a single valid footprint is chosen for a wand that works for each and every potential state satisfying its left-hand side.

4.3.2. Package Logic: Preliminaries

To capture different state models and flavours of separation logic, our package logic is parameterized by the IDF algebra (Definition 2.3.1) presented in Chapter 2. Recall that we write $\succeq$ for the induced partial order of the resulting partial commutative monoid, and $\sigma_1 \# \sigma_2$ iff $\sigma_1 \oplus \sigma_2$ is defined (*i.e.*, σ_1 and σ_2 are compatible). Finally, if $\sigma_2 \succeq \sigma_1$, we define the subtraction $\sigma_2 \ominus \sigma_1$ to be the $\succeq$-largest state σ_r such that $\sigma_2 = \sigma_1 \oplus \sigma_r$.[6]

6: This largest state, which always exists and is unique, can be constructed as follows: Take any state σ such that $\sigma_2 = \sigma_1 \oplus \sigma$. Then $\sigma_r \triangleq \sigma \oplus |\sigma_2|$.

We define our package logic for an assertion language with the following grammar:

$$A ::= A * A \mid \mathcal{B} \Rightarrow A \mid \mathcal{B}$$

where A ranges over assertions and $\mathcal{B}$ over *semantic assertions*.

To allow our package logic to be applied to a variety of underlying assertion logics (including the one from Section 3.2.3), we distinguish only the two most-relevant connectives for our package logic: the separating conjunction and an implication (for expressing conditional assertions). To support additional constructs of the assertion logic, the third type of assertion we consider is a *semantic assertion*, *i.e.*, a function from Σ to Booleans. This third type can be instantiated to represent logical assertions that do not match the first two cases. In particular, assertions such as `x.f = 5`, `acc(x.f)`, recursively-defined predicates (such as `tree(x)`) or magic wands can be represented as semantic assertions. This core assertion language can also be easily extended with native support for *e.g.*, the logical conjunction and disjunction connectives; we explain in the extended version of this chapter [198] how to extend the rules of the logic accordingly.

[198]: Dardinier et al. (2022), *Sound Automation of Magic Wands (Extended Version)*

4.3.3. The Package Logic

We define our package logic to prescribe the design space of algorithms for deciding how, in an initial state σ_0, to select a valid footprint (or fail) for a magic wand $A \mathrel{-\!\!*} B$. The aim is to infer states σ_w and σ_1 that partition σ_0 (*i.e.*, $\sigma_0 = \sigma_1 \oplus \sigma_w$) such that σ_w is a valid footprint for $A \mathrel{-\!\!*} B$ (when combined with any compatible state satisfying A, the resulting state satisfies B). In particular, all permissions (and logical facts) required by the assertion B must either come from the footprint or be guaranteed to be provided by any compatible state satisfying A.

Recall from Section 4.2.3 that the mistake underlying the FIA approach ultimately resulted from allowing multiple different footprints to be selected conditionally on a state satisfying A, rather than a single footprint which works for all such states. Our package logic addresses this concern by defining judgements in terms of the *set* of all states satisfying A; whenever *any* of these tracked states is insufficient to provide a permission

required by B, our logic will force this permission to be added *in general* to the wand's footprint (taken from the current state).

Definition 4.3.1 Witness sets and contexts**.**
*A **witness set** S is a set of pairs of states (σ_A, σ_B); conceptually, the first represents the state available for trying to prove B* in addition *to the current state; this is initially a state satisfying the wand's left-hand side A. The second represents the state assembled (so-far) to attempt to satisfy the right-hand side B. We write S^1 for the set of first elements of all pairs in a witness set S.*

*A **context** Δ is a pair (σ, S) of a state and a witness set; here, σ represents the (as-yet unused remainder of the) current state in which the wand is being packaged.*

The basic idea behind a derivation in our logic is to show how to assemble a witness set in which *all* second elements are states satisfying B, via some combinations of: (1) moving a part of the first element of a pair in the witness set into the second, and (2) moving a part of the outer state σ into *all* first elements of the pairs (this becomes a part of the wand's footprint). The actual judgements of the logic are a little more complex, to correctly record any hypotheses (called *path conditions*) that result from deconstructing conditional assertions in B.

Definition 4.3.2 Configurations and reductions**.**
*A **configuration** represents a current objective in our package logic: the part of the wand's right-hand side still to be satisfied as well as the current state of a footprint computation. A configuration is a triple $\langle B, pc, (\sigma, S)\rangle$, where B is an assertion, pc is a **path condition** (a function from Σ to Booleans), and (σ, S) is a context. Conceptually, B is the assertion still to be satisfied, pc represents hypotheses we are currently working under, and the context (σ, S) tracks the current state and witness set, as described above.*

*A **reduction** is a judgement $\langle B, pc, (\sigma_0, S_0)\rangle \rightsquigarrow (\sigma_1, S_1)$, representing the achievement of the objective described via the configuration on the left, resulting in the final context on the right; σ_1 is the new version of the outer state (and becomes the new current state after the* `package` *operation); whatever was removed from the initial outer state is implicitly the selected footprint state σ_w. If a reduction is derivable in our package logic, this footprint σ_w guarantees that for all $(\sigma_A, \sigma_B) \in S_0$, if $(\sigma_A \oplus \sigma_B)\#\sigma_w$, then $\sigma_A \oplus \sigma_w$ satisfies $pc \Longrightarrow B$. The condition $(\sigma_A \oplus \sigma_B)\#\sigma_w$ ensures that the pair (σ_A, σ_B) actually corresponds to a state in which the wand can be applied given the chosen footprint σ_w, as we explain later. The package logic defines the steps an algorithm may take to achieve this goal.*

We represent packaging a wand $A \mathrel{-\!\!*} B$ in state σ_0 by the derivation of a reduction

$$\langle B, \lambda\sigma.\,\top, (\sigma_0, S_A^{\sigma_0})\rangle \rightsquigarrow (\sigma_1, S_1)$$

for some state σ_1 and witness set S_1, where

$$S_A^{\sigma_0} \triangleq \{(\sigma_A, \mathsf{stabilize}(|\sigma_A|)) \mid \sigma_A \models A \wedge \sigma_A\#\mathsf{stabilize}(|\sigma_0|)\}$$

The path condition is initially true, as we are not yet under any hypotheses. The initial witness set, $S_A^{\sigma_0}$, contains all pairs of a state σ_A that satisfies A

$$\textsc{Implication}\ \frac{\langle A, \lambda\sigma.\, pc(\sigma) \wedge b(\sigma), \Delta\rangle \rightsquigarrow \Delta'}{\langle b \Rightarrow A, pc, \Delta\rangle \rightsquigarrow \Delta'} \qquad \textsc{Star}\ \frac{\langle A_1, pc, \Delta_0\rangle \rightsquigarrow \Delta_1 \qquad \langle A_2, pc, \Delta_1\rangle \rightsquigarrow \Delta_2}{\langle A_1 * A_2, pc, \Delta_0\rangle \rightsquigarrow \Delta_2}$$

$$\textsc{AtomS}\ \frac{\begin{array}{c}\forall(\sigma_A, \sigma_B) \in S.\, pc(\sigma_A) \Longrightarrow \sigma_A \succeq \mathit{choice}(\sigma_A, \sigma_B) \wedge \mathscr{B}(\mathit{choice}(\sigma_A, \sigma_B)) \\ S_\top = \{(\sigma_A \ominus \mathit{choice}(\sigma_A, \sigma_B), \sigma_B \oplus \mathit{choice}(\sigma_A, \sigma_B)) | (\sigma_A, \sigma_B) \in S \wedge pc(\sigma_A)\} \\ S_\bot = \{(\sigma_A, \sigma_B) | (\sigma_A, \sigma_B) \in S \wedge \neg pc(\sigma_A)\}\end{array}}{\langle \mathscr{B}, pc, (\sigma, S)\rangle \rightsquigarrow (\sigma, S_\top \cup S_\bot)}$$

$$\textsc{ExtractS}\ \frac{\mathsf{stable}(\sigma_w) \qquad \langle A, pc, (\sigma_1, S_1)\rangle \rightsquigarrow \Delta \qquad \begin{array}{c}\sigma_0 = \sigma_1 \oplus \sigma_w \\ S_1 = \{(\sigma_A \oplus \sigma_w, \sigma_B) | (\sigma_A, \sigma_B) \in S_0 \wedge (\sigma_A \oplus \sigma_B) \# \sigma_w\}\end{array}}{\langle A, pc, (\sigma_0, S_0)\rangle \rightsquigarrow \Delta}$$

Figure 4.2.: Rules of the package logic, specialized to the standard semantics of magic wands.

and the state $\mathsf{stabilize}(|\sigma_A|)$ (which is the smallest state compatible with σ_A, as shown by the last axiom in Figure 2.6), [7] to which a successful reduction will add permissions in order to satisfy B[8].

An actual algorithm need not explicitly compute this (possibly infinite) set, but can instead track it symbolically. If the algorithm finds a derivation of this reduction, it has proven that the difference between σ_0 and σ_1 is a valid footprint of the wand $A \mathrel{-\!\!*} B$, since the logic is sound (Theorem 4.3.1 below).

Rules of the package logic. Figure 4.2 presents the four rules of our logic, defining (via derivable reductions) how a configuration can be reduced to a context. There is a rule for each type of assertion B: Implication for an implication, Star for a separating conjunction, and AtomS for a semantic assertion. The logic also includes the rule ExtractS, which represents a choice to extract permissions from the outer state and adds them to all pairs of states in the witness set. In the following, we informally write *reducing an assertion* to refer to the process of deriving (in the logic) that the relevant configuration containing this assertion reduces to some context.

To reduce an implication $b \Rightarrow A$, the rule Implication conjoins the hypothesis b with the previous path condition, leaving A to be reduced. Informally, this expresses that satisfying $pc \Rightarrow (b \Rightarrow A)$ is equivalent to satisfying $(pc \wedge b) \Rightarrow A$.

For a separating conjunction $A_1 * A_2$, the rule Star expresses that both A_1 and A_2 must be reduced, in order to reduce $A_1 * A_2$; permissions used in the reduction of the first conjunct must not be used again, which is reflected by the threading-through of the intermediate context Δ_1.[9]

The rule AtomS specifies how to prove that all states in S^1 (where S is the witness set) satisfy the assertion $pc \Rightarrow \mathscr{B}$. To understand the premises, consider a pair $(\sigma_A, \sigma_B) \in S$. If σ_A does not satisfy the path condition (*i.e.*, $\neg pc(\sigma_A)$), then σ_A *does not* have to justify $\mathscr{B}$, and thus the pair (σ_A, σ_B) is left unchanged; this case corresponds to the set $S_\bot$. Conversely, if σ_A satisfies the path condition (*i.e.*, $pc(\sigma_A)$), then σ_A must satisfy $\mathscr{B}$, and the corresponding permissions must be transferred from σ_A to σ_B. Since some assertions may be satisfied in different ways, such as disjunctions,

7: The published version of this chapter [141] used a slightly stronger IDF algebra, which contained a neutral element e (*i.e.*, such that $e \oplus \sigma = \sigma$ for all σ), and thus the initial witness set was defined as $\{(\sigma_A, e) \mid \sigma_A \models A\}$, where e is a neutral element of the IDF algebra (*i.e.*, such that $e \oplus \sigma = \sigma$ for all σ).

8: If B is *affine* (or intuitionistic), this can be simplified to only the $\geq$-*minimal* states that satisfy A. B is *affine* iff for all states σ such that B holds in σ, then B holds in any state σ' such that $\sigma' \geq \sigma$. In affine SL (such as Iris [31]) or in IDF, all assertions are affine.

9: The order in the premises is unimportant since $A_1 * A_2$ and $A_2 * A_1$ are equivalent.

$$\textsc{Star}\ \frac{\textsc{Atom}\ \dfrac{\ldots}{\langle \mathbf{acc}(x.f), (\sigma_0, S_0)\rangle \rightsquigarrow (\sigma_0, S_1)} \qquad \textsc{Extract}\ \dfrac{\textsc{Atom}\ \dfrac{\ldots}{\langle \mathbf{acc}(x.f.g), (\sigma_1, S_2)\rangle \rightsquigarrow (\sigma_1, S_3)}\ \dagger}{\langle \mathbf{acc}(x.f.g), (\sigma_0, S_1)\rangle \rightsquigarrow (\sigma_1, S_3)}}{\langle \mathbf{acc}(x.f) * \mathbf{acc}(x.f.g), (\sigma_0, S_0)\rangle \rightsquigarrow (\sigma_1, S_3)}$$

Figure 4.3.: Example of a (part of a) derivation in the package logic. This derivation shows how the rules from the package logic can be used to package the wand from Example 4.3.1, $\mathbf{acc}(x.b, 1/2) \mathrel{-\!\!*} \mathbf{acc}(x.b, 1/2) * (x.b \Rightarrow \mathbf{acc}(x.f))$. We omit the path condition since it is always the trivial condition ($\lambda\sigma.\ \top$). Assume that the outer state σ_0 is the addition of σ_{yz}, a state that contains permission to `y.g` and `z.g`, and σ_1. $S_0 \triangleq \{(\sigma_A, \mathsf{stabilize}(|\sigma_A|)) \mid \sigma_A \in \Sigma \wedge \sigma_A \models \mathbf{acc}(x.f) * (x.f = y \vee x.f = z)\}$ is the initial witness set. The derivation shows that $\langle \mathbf{acc}(x.f) * acc(x.f.g), (\sigma_0, S_0)\rangle \rightsquigarrow (\sigma_1, S_3)$ is correct, and thus that σ_{yz} is a correct footprint of the wand w (since $\sigma_0 = \sigma_1 \oplus \sigma_{yz}$).

the algorithm has a choice in how to satisfy $\mathcal{B}$, which might be different for each pair (σ_A, σ_B). This choice is represented by $choice(\sigma_A, \sigma_B)$, which must satisfy $\mathcal{B}$ and be smaller or equal to σ_A. We update the witness set by transferring $choice(\sigma_A, \sigma_B)$ from σ_A to σ_B. This second case corresponds to the set $S_\top$. Note that the rule AtomS can be applied only if σ_A satisfies $\mathcal{B}$, for all pairs $(\sigma_A, \sigma_B) \in S$ such that $pc(\sigma_A)$. If not, a package algorithm must either first extract more permissions from the outer state with the rule ExtractS, or fail.

Choice in the rule AtomS and angelism in the semantics of CoreIVL

Interestingly, our use of a choice function (*choice*) in the rule AtomS is very similar to how we use a choice function $\mathcal{S}$ in the rule SeqOp from Figure 2.7, to model angelism in the semantics of CoreIVL. The two choice functions capture the fact that a potentially infinite number of states must make *individual* choices: How to resolve angelic choices in the rule SeqOp, and how to satisfy the assertion $\mathcal{B}$ in the rule AtomS. Those choice functions also make it necessary to use the axiom of choice to prove completeness of the corresponding derivation system (Theorem 2.3.2 for the semantics of CoreIVL, and Theorem 4.3.2 for the package logic).

The rule ExtractS (applicable at any step of a derivation), expresses that we can extract permissions (the state σ_w, required to be stable[10]) from the outer state σ_0, and combine them with the first element of each pair of states in the witness set. Note that (σ_A, σ_B) is removed from the witness set if $\sigma_A \oplus \sigma_B$ is not compatible with σ_w. In such cases, adding σ_w to σ_A would create a pair in the witness set representing a state in which the wand cannot be applied. Consequently, there is no need to establish the right-hand side of the wand for this pair and our logic correspondingly removes it. Finally, the rule requires that we reduce the assertion A in the new context.

10: Recall that in standard SL, all states are stable.

A package algorithm's strategy is mostly reflected by how it uses the rule ExtractS. To package the wand $\mathbf{acc}(x.b, 1/2) \mathrel{-\!\!*} \mathbf{acc}(x.b, 1/2) * (x.b \Rightarrow \mathbf{acc}(x.f))$ from Example 4.3.1 one algorithm might use this rule to extract permission to `x.f`; another might use it to extract permission to `x.b` (if `x.b` had value false in the original state).

Example 4.3.2 Example of a derivation in the package logic.

Figure 4.3 illustrates how the rules of our package logic can be used to package the wand from Section 4.3.1, $\mathbf{acc}(x.b, 1/2) \mathrel{-\!\!*} \mathbf{acc}(x.b, 1/2) * (x.b \implies \mathbf{acc}(x.f))$. This derivation, which reflects the package algorithm that we will describe in Section 4.3.5, can be read from bottom to top and from left to right. Using the rule Star, we split the assertion into its two conjuncts, $\mathbf{acc}(x.f)$ (on the left) and $\mathbf{acc}(x.f.g)$ (on the right). We then handle $\mathbf{acc}(x.f)$ using the rule AtomS. $\mathbf{acc}(x.f)$ holds in the first element of each pair of S_0, since any state that satisfies the wand's left-hand side owns `x.f`. Therefore, we use the rule AtomS with a *choice* function that always chooses the relevant state with exactly full permission to `x.f`. S_1 is the updated witness set where this permission to `x.f` has been transferred from the first to the second element of each pair of states.

Next, we handle $\mathbf{acc}(x.f.g)$. We cannot do this directly using the rule AtomS from S_1. We know that, for each $(\sigma_A, \sigma_B) \in S_1$, `x.f.g` evaluated in σ_A is either `y` or `z`, but σ_A owns neither `y.g` nor `z.g`. So, we transfer the permissions to both `y.g` and `z.g` from the outer state σ_0 to all states of S_1^1, using the rule ExtractS, which results in the context (σ_1, S_2); † represents the three other premises of the rule, namely $\sigma_0 = \sigma_{yz} \oplus \sigma_1$, $\mathsf{stable}(\sigma_{yz})$, and S_2's definition. Finally, we apply the rule AtomS to prove $\langle \mathbf{acc}(x.f.g), (\sigma_1, S_2)\rangle \rightsquigarrow (\sigma_1, S_3)$, where the *choice* function chooses for each pair the corresponding state that contains full permission to `x.f.g`.

4.3.4. Soundness and Completeness

We write $\vdash \langle B, pc, \Delta\rangle \rightsquigarrow \Delta'$ to express that a reduction can be derived in the logic. As explained above, the goal of a package algorithm is to find a derivation of $\langle B, \lambda\_.\ \top, (\sigma, \{(\sigma_A, \mathsf{stabilize}(|\sigma_A|)) \mid \sigma_A \in S_A\})\rangle \rightsquigarrow (\sigma', S')$. If it succeeds, then the difference between σ' and σ is a valid footprint of $A \mathrel{-\!\!*} B$, since our package logic is sound. In particular, we have proven the following soundness result in Isabelle/HOL:

Theorem 4.3.1 Soundness of the package logic**.**
Let $S_A^\sigma \triangleq \{(\sigma_A, \mathit{stabilize}(|\sigma_A|)) \mid \sigma_A \models A \wedge \sigma_A \#\mathit{stabilize}(|\sigma|)\}$*, and* B *be a well-formed*[11] *assertion. If*

$$\vdash \langle B, \lambda\_.\ \top, (\sigma, S_A^\sigma)\rangle \rightsquigarrow (\sigma', S')$$

and at least one of the following conditions holds:

1. B *is affine, or*
2. *for all* $(\sigma_A, \sigma_B) \in S'$, σ_A *contains no permission* (i.e., $\sigma_A \oplus \sigma_A = \sigma_A$)

then there exists a stable state σ_w *s.t.* $\sigma = \sigma' \oplus \sigma_w$ *and* $\sigma_w \models A \mathrel{-\!\!*} B$.

11: Intuitively, an assertion A is well-formed iff A is self-framing and all implications in A have pure left-hand sides.

The third premise shows that, in an affine SL or IDF setting, the correspondence between a derivation in the logic and a valid footprint of a wand is straightforward (case 1). However, in classical SL, one must additionally check that all permissions in the witness set have been consumed (case 2).

We have also proved in Isabelle/HOL that our package logic is complete, *i.e.*, *any* valid footprint can be computed via a derivation in our package

logic:

Theorem 4.3.2 Completeness of the package logic.
Let $S_A^\sigma \triangleq \{(\sigma_A, \mathit{stabilize}(|\sigma_A|)) \mid \sigma_A \models A \wedge \sigma_A \#\mathit{stabilize}(|\sigma|)\}$, *and* B *be a well-formed assertion. If* σ_w *is a stable footprint of* $A \mathrel{-\!\!*} B$, *and* $\sigma = \sigma' \oplus \sigma_w$, *then there exists a witness set* S' *such that*

$$\vdash \langle B, \lambda\_.\, \top, (\sigma, S_A^\sigma) \mid \sigma_A \models A\}) \rangle \rightsquigarrow (\sigma', S')$$

4.3.5. A Sound Package Algorithm

We now describe an automatic package algorithm that corresponds to a proof search strategy in our package logic, and which is thus sound. To convey the main ideas, consider packaging a wand of the shape $A \mathrel{-\!\!*} B_1 * \ldots * B_n$.[12] Our algorithm traverses the assertion $B_1 * \ldots * B_n$ from left to right, similarly to the FIA approach; this traversal is justified by repeated applications of the rule Star. Assume at some point during this traversal that the current context is (σ_0, S). When we encounter the assertion B_i, we have two possible cases:

1. All states $\sigma_A \in S^1$ satisfy B_i, which means that the permissions (or values) required by B_i are provided by the left-hand side of the wand. In this case, for each pair $(\sigma_A, \sigma_B) \in S$, we transfer permissions (and the corresponding values) to satisfy B_i from σ_A to σ_B, using the rule AtomS. Note that the transferred permissions might be different for each pair (σ_A, σ_B). This gives us a new witness set S', while the outer state σ_0 is left unchanged. We must then handle the next assertion B_{i+1} in the context (σ_0, S').
2. There is at least one pair $(\sigma_A, \sigma_B) \in S$ such that B_i does not hold in σ_A. In this case, the algorithm fails if combining the permissions (and values) contained in the outer state with each $\sigma_A \in S^1$ is not sufficient to satisfy B_i. Otherwise, we apply the rule ExtractS to transfer permissions from the outer state σ_0 to each state σ_A in S^1 such that B_i holds in σ_A. This gives us a new context (σ_0', S'). We can now apply the first case with the context (σ_0', S').

12: In the extended version of this chapter [141], we present a more formal version of our package algorithm, and also show how it handles implications.

4.4. Generalized Package Logic

The previous section presented a version of our package logic *specialized* to the standard semantics of magic wands. In this section, we present the *generalized* version of the package logic, which allows packaging magic wands with alternative semantics, for example to obtain wands that are combinable (as defined in Section 3.3).

We first describe the generalization of the package logic (Section 4.4.1) for a novel parametric definition of magic wands (which allows for alternative semantics), and then introduce a novel definition of *combinable wands*[13] (Section 4.4.2) that can be packaged using the generalized package logic. We will see in Section 4.5 that the restriction combinable wands impose is sufficiently weak for practical purposes, and that footprints of combinable wands can be automatically inferred by package algorithms

13: This novel definition of *combinable wands* is different from the one based on the unbounded logic from Chapter 3, as we explain in Section 4.4.2. The *unbounded wand* from Chapter 3 can be supported by the specialized version of the package logic presented in Section 4.3, as the difference with the *standard wand* is only in the state model, not in the definition of validity.

built on our package logic. All results in this section have been proven in Isabelle/HOL.

4.4.1. Generalizing the Logic

The generalization of the package logic allows us to package *parametric magic wands* $A \mathrel{-\!\!*}_{\mathcal{T}} B$, whose semantics transform the footprint via *monotonic transformers* before combining it with states that satisfy A. We will show, in Section 4.4.2, how to instantiate the parameter $\mathcal{T}$ to obtain combinable wands.

Definition 4.4.1 Parametric magic wands.
*A **monotonic transformer** τ is a function from Σ to Σ (i.e., from states to states) that is monotonic and maps empty states to empty states, i.e., it satisfies the following properties:*

1. $\forall \sigma.\, \tau(\mathit{stabilize}(|\sigma|)) = \mathit{stabilize}(|\sigma|)$
2. $\forall \sigma_1, \sigma_2.\, \sigma_2 \succeq \sigma_1 \implies \tau(\sigma_2) \succeq \tau(\sigma_1)$

*Given a family $(\mathcal{T}_\sigma)_{\sigma \in \Sigma}$ of monotonic transformer indexed by states, the semantics of the **parametric magic wand** $A \mathrel{-\!\!*}_{\mathcal{T}} B$ applies the transformer $\mathcal{T}_\sigma$ to the footprint σ_F before combining it with a state σ that satisfies the left-hand side A. Formally, a state σ_F satisfies the parametric magic wand $A \mathrel{-\!\!*}_{\mathcal{T}} B$, written $\sigma_F \models A \mathrel{-\!\!*}_{\mathcal{T}} B$, if and only if the following holds:*

$$\forall \sigma.\, \sigma \models A \land \sigma \# \mathcal{T}_\sigma(\sigma_F) \implies \sigma \oplus \mathcal{T}_\sigma(\sigma_F) \models B$$

Example 4.4.1 The standard wand is a parametric wand.
Let $\mathcal{I}$ be a family of identity transformers, *i.e.*, for all σ and σ_F, $\mathcal{I}_\sigma(\sigma_F) = \sigma_F$. Note that $\mathcal{I}$ is trivially a monotonic transformer.

The parametric magic wand $A \mathrel{-\!\!*}_{\mathcal{I}} B$ is equivalent to the standard magic wand $A \mathrel{-\!\!*} B$.

We will see another example of monotonic transformer in Section 4.4.2, where we define a family $\mathcal{R}$ of monotonic transformers to ensure that fractions of different footprints combine to valid footprints.

To obtain the generalized version of our package logic, which allows packaging parametric wands, we first generalize witness sets and configurations as follows, where the changes from the specialized package logic are highlighted in blue:

Definition 4.4.2 Generalized witness sets and contexts.
*A **generalized witness set** S is a set of triples $(\sigma_A, \sigma_B, \tau)$, where σ_A and σ_B are states, and τ is a monotonic transformer.*

*A **generalized context** is a triple (σ, σ_F, S), where σ is a state, σ_F is the footprint extracted so far, and S is a generalized witness set.*

Configurations and reductions in the generalized package logic use generalized witness sets and contexts, but otherwise are left unchanged. To package the generalized magic wand $A \mathrel{-\!\!*}_{\mathcal{T}} B$, we need to find a

$$\textsc{Atom}\ \frac{\begin{array}{c}\forall(\sigma_A,\sigma_B,\tau)\in S.\, pc(\sigma_A)\implies \sigma_A \geq choice(\sigma_A,\sigma_B,\tau)\wedge \mathcal{B}(choice(\sigma_A,\sigma_B,\tau))\\ S_\top=\{(\sigma_A\ominus choice(\sigma_A,\sigma_B,\tau),\sigma_B\oplus choice(\sigma_A,\sigma_B),\tau)|(\sigma_A,\sigma_B,\tau)\in S\wedge pc(\sigma_A)\}\\ S_\bot=\{(\sigma_A,\sigma_B,\tau)|(\sigma_A,\sigma_B,\tau)\in S\wedge\neg pc(\sigma_A)\}\end{array}}{\langle\mathcal{B},pc,(\sigma,\sigma_F,S)\rangle\rightsquigarrow(\sigma,\sigma_F,S_\top\cup S_\bot)}$$

$$\textsc{Extract}\ \frac{\sigma_0=\sigma_1\oplus\sigma_E\qquad \mathsf{stable}(\sigma_E)\qquad \langle A,pc,(\sigma_1,\sigma_F\oplus\sigma_E,S_1)\rangle\rightsquigarrow\Delta\qquad S_1=\{(\sigma_A\oplus(\tau(\sigma_F\oplus\sigma_E)\ominus\tau(\sigma_F)),\sigma_B,\tau)|(\sigma_A,\sigma_B,\tau)\in S_0\wedge(\sigma_A\oplus\sigma_B)\#\sigma_E\}}{\langle A,pc,(\sigma_0,\sigma_F,S_1)\rangle\rightsquigarrow\Delta}$$

Figure 4.4.: Rules of the generalized package logic. The changes from the specialized package logic are highlighted in blue. The rules for implication and separating conjunctions (Implication and Star) are unchanged from the specialized package logic.

derivation, in the generalized package logic, of the following reduction:

$$\langle B,\top,(\sigma_0,\mathsf{stabilize}(|\sigma_0|),S_A^{\sigma_0})\rangle\rightsquigarrow(\sigma_1,\sigma_F,S_1)$$

where

$$S_A^{\sigma_0}\triangleq\{(\sigma,\mathsf{stabilize}(|\sigma|),\mathcal{T}_\sigma)\mid\sigma\models A\wedge\sigma\#\mathsf{stabilize}(|\sigma_0|)\}$$

and $\mathsf{stabilize}(|\sigma_0|)$ and $\mathsf{stabilize}(|\sigma|)$ are units for the states σ_0 and σ, respectively.

We show in Figure 4.4 the generalized rules Atom and Extract (the rules Implication and Star are unchanged from the specialized package logic). The rule Atom is very similar to the rule AtomS from Figure 4.2, except that it now uses generalized witness sets and contexts.

The generalized rule Extract is more involved: It now combines a *transformed* version of σ_E (via the corresponding transformer) to elements of the witness set (recall that σ_E represents the permissions we extract from the outer state). Crucially, we add (a) $\tau(\sigma_F\oplus\sigma_E)\ominus\tau(\sigma_F)$ instead of the simpler (b) $\tau(\sigma_E)$ to σ_A, because the footprint might be extracted from the outer state σ_0 in a piecewise manner, with several applications of the rule Extract, whereas the definition of the parametric magic wand (Definition 4.4.1) requires the transformer to be applied to the *complete* footprint. Consider for example extracting the footprint $\sigma_F\triangleq\sigma_{E_1}\oplus\sigma_{E_2}$ from the outer state via two successive applications of the rule Extract, first for σ_{E_1} and then for σ_{E_2}. Given an outer state σ and a triple (σ_A,σ_B,τ) from the initial witness set, we update the first element to $\sigma_A\oplus(\tau(\sigma_{E_1})\ominus\tau(\mathsf{stabilize}(|\sigma|)))=\sigma_A\oplus\tau(\sigma_{E_1})$ via the first rule application,[14] and then to $(\sigma_A\oplus\tau(\sigma_{E_1}))\oplus(\tau(\sigma_{E_1}\oplus\sigma_{E_2})\ominus\tau(\sigma_{E_1}))=\sigma_A\oplus\tau(\sigma_{E_1}\oplus\sigma_{E_2})$ via the second rule application. If we followed (b), we would obtain $\sigma_A\oplus\tau(\sigma_{E_1})\oplus\tau(\sigma_{E_2})$ instead, which might be different from $\sigma_A\oplus\tau(\sigma_{E_1}\oplus\sigma_{E_2})$. This illustrates why we need to track the footprint extracted so far in the generalized package logic. Finally, the current footprint is updated to be $\sigma_F\oplus\sigma_E$.

14: Since σ_{E_1} is extracted from the outer state σ, we have $\sigma_{E_1}\geq\mathsf{stabilize}(|\sigma|)$, and thus $\tau(\sigma_{E_1})\geq\tau(\mathsf{stabilize}(|\sigma|))=\mathsf{stabilize}(|\sigma|)$ by monotonicity of τ, which implies $\tau(\sigma_{E_1})\ominus\tau(\mathsf{stabilize}(|\sigma|))=\tau(\sigma_{E_1})$.

We have proven in Isabelle/HOL that this generalized logic is sound and complete for parametric wands:

Theorem 4.4.1 Soundness of the generalized package logic.
Let $S_A^{\sigma_0}\triangleq\{(\sigma,\mathsf{stabilize}(|\sigma|),\mathcal{T}_\sigma)\mid\sigma\models A\wedge\sigma\#\mathsf{stabilize}(|\sigma_0|)\}$, B *be a*

well-formed assertion, and $(\mathcal{T}_\sigma)_{\sigma\in\Sigma}$ *be a family of monotonic transformers. If*

$$\vdash \langle B, \top, (\sigma_0, \mathit{stabilize}(|\sigma_0|), S_A^{\sigma_0})\rangle \rightsquigarrow (\sigma_1, \sigma_F, S_1)$$

for some witness set S_1*, and states* σ_1 *and* σ_F*, and at least one of the following conditions holds:*

1. *B is affine, or*
2. *for all* $(\sigma_A, \sigma_B) \in S_1$, σ_A *contains no permission* (i.e., $\sigma_A \oplus \sigma_A = \sigma_A$)

then σ_F *is a stable footprint of* $A \mathrel{-\!\!*}_{\mathcal{T}} B$*, and* $\sigma_0 = \sigma_1 \oplus \sigma_F$.

Theorem 4.4.2 Completeness of the generalized package logic.
Let $S_A^{\sigma_0} \triangleq \{(\sigma, \mathit{stabilize}(|\sigma|), \mathcal{T}_\sigma) \mid \sigma \models A \wedge \sigma\#\mathit{stabilize}(|\sigma_0|)\}$*, B be a well-formed assertion, and* $(\mathcal{T}_\sigma)_{\sigma\in\Sigma}$ *be a family of monotonic transformers. If* σ_F *is a stable footprint of* $A \mathrel{-\!\!*}_{\mathcal{T}} B$*, and* $\sigma_0 = \sigma_1 \oplus \sigma_F$*, then there exists a witness set* S_1 *such that*

$$\vdash \langle B, \top, (\sigma_0, \mathit{stabilize}(|\sigma_0|), S_A^{\sigma_0})\rangle \rightsquigarrow (\sigma_1, \sigma_F, S_1)$$

4.4.2. Using the Generalized Package Logic with Combinable Wands

We now introduce a novel notion of *combinable wand*, written $A \mathrel{-\!\!*}_c B$, which can be expressed as a parametric magic wand, and thus can be packaged using the generalized package logic. Unlike the *standard wand* $A \mathrel{-\!\!*} B$, the combinable wand $A \mathrel{-\!\!*}_c B$ is combinable if B is combinable (Definition 3.3.1). Note that the novel definition of *combinable wand* we present here is different from the one based on the unbounded logic from Chapter 3, which we refer to as *unbounded wand*. Combinable and unbounded wands strike a different trade-off: The unbounded wand is in general more restrictive than the combinable wand, but it additionally enjoys distributivity (as shown by the rule DotWand in Figure 3.4). Moreover, the unbounded wand can be packaged using the specialized package logic presented in Section 4.3, as the difference with the standard wand is only in the state model, not in the definition of validity. In contrast, the combinable wand differs from the standard wand in the definition of validity, and thus requires the generalized package logic presented in Section 4.4.1.

To explain the intuition behind combinable wands, let us first recall why the standard wand $A \mathrel{-\!\!*} B$ is not combinable in general, with the following example:

Example 4.4.2 A simple non-combinable wand.
The wand

$$W \triangleq \mathbf{acc}(x.f, 1/2) \mathrel{-\!\!*} \mathbf{acc}(x.g)$$

is not combinable, because $0.5 \cdot W * 0.5 \cdot W \not\models W$. To see this, consider two states σ_F and σ_G, containing full permissions to only `x.f` and `x.g`, respectively. Both states are valid footprints of W, *i.e.*, $\sigma_F \models w$ (because σ_F is incompatible with all states that satisfy the left-hand side) and $\sigma_G \models W$ (because σ_G entails the right-hand side). Thus, by definition

(Figure 3.3), $0.5 \otimes \sigma_F \models 0.5 \cdot W$ and $0.5 \otimes \sigma_G \models 0.5 \cdot W$. However, $0.5 \otimes \sigma_F \oplus 0.5 \otimes \sigma_G$, *i.e.*, a state with half permission to both `x.f` and `x.g`, is *not* a valid footprint of W, and thus $0.5 \cdot W * 0.5 \cdot W \not\models W$.

Intuitively, W is not combinable because one of its footprints, σ_F, is incompatible with the left-hand side of the wand, but becomes compatible when the footprint is scaled down to a fraction. After scaling, the wand no longer holds trivially, and the footprint does not necessarily establish the right-hand side.

To make this intuition more precise, we introduce the notion of *scalable footprints*:

Definition 4.4.3 Scalable footprints**.**
For a state σ, we define scaled(σ) to be the set of copies of σ multiplied by any fraction $0 < \alpha \leq 1$, i.e., *scaled*$(\sigma) \triangleq \{\alpha \otimes \sigma \mid 0 < \alpha \leq 1\}$.

*A footprint σ_E is **scalable w.r.t. a state** σ_A iff either*

1. *σ_A is compatible with* all *states from scaled(σ_E), or*
2. *σ_A is compatible with* no *state in scaled(σ_E).*

*A footprint is **scalable for a wand** $A \mathrel{-\!\!*} B$ iff it is scalable w.r.t. all states that satisfy A.*

Intuitively, this means that the footprint does not "jump" between satisfying the wand trivially and having to satisfy the right-hand side. In the above example, σ_G is a scalable footprint for w, but σ_F is not.

Making wands combinable. The previous paragraphs show that, even if B is combinable, the standard wand $A \mathrel{-\!\!*} B$ is in general not combinable because it can be satisfied by non-scalable footprints. Therefore, we define the validity of *combinable wands* $A \mathrel{-\!\!*}_c B$ to *force* footprints to be scalable, in the following sense. A combinable wand accepts all scalable footprints, and transforms (via a *monotonic transformer*) non-scalable footprints before checking whether they actually satisfy the wand. We obtain the definition of combinable wands by replacing a potential footprint σ_F with a (possibly smaller) state $\mathcal{R}_{\sigma_A}(\sigma_F)$ that is scalable w.r.t. σ_A. $\mathcal{R}_{\sigma_A}(\sigma_F)$ is defined as σ_F if *no* state in *scaled*(σ_F) is compatible with σ_A; in that case, condition (2) of scalable footprints holds for $\mathcal{R}_{\sigma_A}(\sigma_F)$ w.r.t. σ_A. Otherwise, $\mathcal{R}_{\sigma_A}(\sigma_F)$ is obtained by removing just enough permissions from σ_F to ensure that *all* states in *scaled*$(\mathcal{R}_{\sigma_A}(\sigma_F))$ are compatible with σ_A, which ensures that condition (1) holds for $\mathcal{R}_{\sigma_A}(\sigma_F)$ w.r.t. σ_A.

To formally define $\mathcal{R}_{\sigma_A}(\sigma_F)$, we fix the IDF algebra Σ_{IDF} from Section 2.3.1, whose states are pairs (π, h) of a (bounded) *permission mask* π, which maps heap locations to fractional permissions, and a partial heap h, which maps heap locations to values.

Definition 4.4.4 Combinable wands**.**
Let $\sigma_A \triangleq (\pi_A, h_A)$ and $\sigma_F \triangleq (\pi_F, h_F)$ be states from Σ_{IDF}, and let π'_F be the permission mask such that $\forall l.\ \pi'_F(l) = \min(\pi_F(l), 1 - \pi_A(l))$. The family of

monotonic transformers $\mathcal{R}$ is defined as follows:

$$\mathcal{R}_{\sigma_A}(\sigma_F) \triangleq \begin{cases} \sigma_F & \text{if } \forall \sigma \in \mathit{scaled}(\sigma_F).\ \neg \sigma_A \# \sigma \\ (\pi'_F, h_F) & \text{otherwise} \end{cases}$$

The combinable wand *$A \mathrel{-\!\!*}_c B$ is then interpreted as a parametric magic wand (Definition 4.4.1) with the family of monotonic transformers $\mathcal{R}$, as follows:*

$$\sigma_F \models A \mathrel{-\!\!*}_c B \ \triangleq\ \left(\forall \sigma_A.\ \sigma_A \models A \wedge \sigma_A \# \mathcal{R}_{\sigma_A}(\sigma_F) \Rightarrow \sigma_A \oplus \mathcal{R}_{\sigma_A}(\sigma_F) \models B\right)$$

The following proposition (proved in Isabelle/HOL) shows some key properties of combinable wands.

Proposition 4.4.3 Key properties of combinable wands**.**
Let B be an affine assertion.

1. *If B is combinable, then $A \mathrel{-\!\!*}_c B$ is combinable.*
2. *$A \mathrel{-\!\!*}_c B \models A \mathrel{-\!\!*} B$.*
3. *If A is a* binary *assertion, then $A \mathrel{-\!\!*}_c B$ and $A \mathrel{-\!\!*} B$ are equivalent.*

Property 1 expresses that combinable wands constructed from combinable assertions are combinable, which enables verification methodologies underlying tools such as VerCors and Viper to support flexible combinations of wands and predicates (as motivated in Chapter 3). Property 2 implies that $A * (A \mathrel{-\!\!*}_c B) \models B$, that is, combinable wands can be applied like standard wands. Property 3 states that combinable wands pose no restrictions if the left-hand side is binary, that is, if it can be expressed without fractional permissions[15] For example, the predicate $\mathit{tree}(x)$ from Figure 4.1 is binary, which implies that the wands $\mathit{tree}(y) \mathrel{-\!\!*}_c \mathit{tree}(x)$ and $\mathit{tree}(y) \mathrel{-\!\!*} \mathit{tree}(x)$ are equivalent. This property is an important reason for why combinable wands are expressive enough for practical purposes, as we further evidence in Section 4.5.

15: Formally, an assertion A is *binary* iff $\forall(\pi, h) \models A.\ (\mathit{bin}(\pi), h) \models A$, where $\mathit{bin}(\pi)$ is the *binary restriction* of the permission mask π, defined as $\mathit{bin}(\pi)(l) \triangleq \begin{cases} 1 & \text{if } \pi(l) = 1 \\ 0 & \text{otherwise} \end{cases}$.

Example 4.4.3 A combinable wand**.**
Consider again the standard wand $W \triangleq \mathbf{acc}(x.f, 1/2) \mathrel{-\!\!*} \mathbf{acc}(x.g)$ and the states σ_F and σ_G, containing full permissions to only `x.f` and `x.g`, respectively. As explained in Section 4.4, w is not combinable, because it holds in both σ_F and σ_G, but not in $0.5 \otimes \sigma_F \oplus 0.5 \otimes \sigma_G$.

Consider now the combinable wand $W_c \triangleq \mathbf{acc}(x.f, 1/2) \mathrel{-\!\!*}_c \mathbf{acc}(x.g)$, which is combinable because $\mathbf{acc}(x.g)$ is combinable (Theorem 4.4.3). σ_G is a valid footprint of W_c. To see this, consider a state σ_A in which the left-hand side $\mathbf{acc}(x.f, 1/2)$ holds, and with no permission to `x.g`. By Definition 4.4.4, $\mathcal{R}_{\sigma_A}(\sigma_G) = \sigma_G$, and thus $\sigma_A \oplus \mathcal{R}_{\sigma_A}(\sigma_G) = \sigma_A \oplus \sigma_G \models \mathbf{acc}(x.g)$.

In contrast, σ_F is *not* a footprint of W_c. Indeed, consider a state σ_A with the same value as σ_F for `x.f` and in which $\mathbf{acc}(x.f, 1/2)$ holds. Since σ_A is compatible with $0.5 \otimes \sigma_F$, the second case of the definition of $\mathcal{R}$ (Definition 4.4.4) applies, and thus $\mathcal{R}_{\sigma_A}(\sigma_F)$ only has 0.5 ($\min(1, 0.5)$) permission to `x.f`. Therefore, $\sigma_A \oplus \mathcal{R}_{\sigma_A}(\sigma_F)$ is defined, but does not satisfy $\mathbf{acc}(x.g)$, and thus W_c does not hold in σ_F.

Standard, unbounded, and combinable wands

We have seen so far three different (meanings for) magic wands, the *standard wand*, the *unbounded wand* (Section 3.2), and the *combinable wand* (Definition 4.4.4), which we compare in the following table:

Wand type	Standard	Unbounded	Combinable
Semantics	Standard	Standard	Definition 4.4.4
State model	Bounded	Unbounded	Bounded
Combinability	No	Yes	Yes
Distributivity	No	Yes	No

This table first shows that unbounded wands have the same definition as standard wands, but they are interpreted in an *unbounded* state model. This allows unbounded wands to be packaged using the specialized package logic from Section 4.3. In contrast, combinable wands differ from standard wands in their definition of satisfiability, and thus they require the generalized package logic from Section 4.4.1. Interestingly, the fourth combination (interpreting Definition 4.4.4 in the unbounded state model) leads to the same semantics as unbounded wands, as all footprints are scalable in the unbounded state model. Definition 4.4.4 can thus be seen as an approximation of the semantics of unbounded wands in the bounded state model.

Second, this table shows how they differ in their properties. Both unbounded and combinable wands strengthen the definition of standard wands to allow combinability. However, only the unbounded wand satisfies distributivity. Interestingly, we cannot obtain a wand definition that is distributive but not combinable, as distributivity implies combinability.[16] To see why the combinable wand is not distributive, consider the combinable wand $\mathbf{acc}(x.f) \mathrel{-\!\!*_c} \mathbf{acc}(y.g)$. This wand is equivalent to the standard wand $\mathbf{acc}(x.f) \mathrel{-\!\!*} \mathbf{acc}(y.g)$ (because of property 3 from Proposition 4.4.3), and thus is satisfied by any state σ with some non-zero permission to `x.f`, for example 0.5. However, $0.5 \otimes \sigma$, which satisfies $0.5 \cdot \mathbf{acc}(x.f) \mathrel{-\!\!*_c} \mathbf{acc}(y.g)$ by definition, does not satisfy $\mathbf{acc}(x.f, 0.5) \mathrel{-\!\!*_c} \mathbf{acc}(y.g, 0.5)$, as it only contains 0.25 permission to `x.f`.

Moreover, the combinable and unbounded wands are *incomparable*. The combinable wand is not stronger than the unbounded wand, as the state σ described above satisfies $\mathbf{acc}(x.f) \mathrel{-\!\!*_c} \mathbf{acc}(y.g)$, but not the unbounded wand $\mathbf{acc}(x.f) \mathrel{-\!\!*} \mathbf{acc}(y.g)$. Conversely, and perhaps surprisingly, the unbounded wand is not stronger than the combinable wand, as illustrated by the following magic wand:

$$W \triangleq \mathbf{acc}(x.b) * (x.b \Rightarrow \mathbf{acc}(x.f, 0.5)) \mathrel{-\!\!*} \mathbf{acc}(x.b) * (x.b \Rightarrow \mathbf{acc}(x.f, 1.5))$$

Interpreted in the unbounded state model, this unbounded wand is satisfied by a state σ with full permission to `x.f` (the full permission matters when combined with a state in which `x.b` holds). In contrast, the state σ does not satisfy the corresponding combinable wand, as the right-hand side in this case is equivalent to $\mathbf{acc}(x.b) * (x.b \Rightarrow \bot)$

16: To see this, take an arbitrary wand $\mathrel{-\!\!*}$ (*i.e.*, that satisfies $A * (A \mathrel{-\!\!*} B) \models B$), which also satisfies distributivity, and α and β such that $\alpha + \beta = 1$. Now take a state σ satisfying $\alpha \cdot (A \mathrel{-\!\!*} B) * \beta \cdot (A \mathrel{-\!\!*} B)$. By distributivity and definition of the star, we get σ_1 and σ_2 such that $\sigma = \sigma_1 \oplus \sigma_2$, where (1) σ_1 satisfies $(\alpha \cdot A) \mathrel{-\!\!*} (\alpha \cdot B)$ and (2) σ_2 satisfies $(\beta \cdot A) \mathrel{-\!\!*} (\beta \cdot B)$. Now take a state σ_A satisfying A and such that $\sigma \# \sigma_A$. We have $\sigma \oplus \sigma_A = (\sigma_1 \oplus \alpha \otimes \sigma_A) \oplus (\sigma_2 \oplus \beta \otimes \sigma_A)$, and $\sigma_1 \oplus \alpha \otimes \sigma_A$ satisfies $\alpha \cdot B$ by (1), and $\sigma_2 \oplus \beta \otimes \sigma_A$ satisfies $\beta \cdot B$ by (2), thus σ satisfies $(\alpha \cdot B) * (\beta \cdot B)$. If B is combinable, then σ satisfies $(\alpha + \beta) \cdot B = B$, and thus $\alpha \cdot (A \mathrel{-\!\!*} B) * \beta \cdot (A \mathrel{-\!\!*} B) \models A \mathrel{-\!\!*} B$, which is the definition of combinability.

(because of the *bounded* state model). Thus, a combination of σ with a state in which `x.b` holds does not satisfy the right-hand side.

While unbounded and combinable wands are incomparable *in theory*, the combinable wand is often less restrictive *in practice*, in particular because of property 3 from Proposition 4.4.3. Thus, which type of wand to use in practice depends on the use case and properties needed (combinability, distributivity, or neither). Nonetheless, our package logic supports the three types.

4.5. Evaluation

We have implemented package algorithms for the standard wands and combinable wands in a custom branch of Viper's [16] verification condition generator (VCG). The two implementations are based on the package logic described in Section 4.3 and Section 4.4, and on the proof search strategy outlined in Section 4.3.5. Viper's VCG translates Viper programs to Boogie [12] programs. It uses a total-heap semantics of IDF [74, 87], where Viper states include a heap and a permission mask (tracking fractional permission amounts). The heap and mask are represented in Boogie as maps; we also represent witness sets as Boogie maps.

[16]: Müller et al. (2016), *Viper*

[12]: Leino (2008), *This Is Boogie 2*

[74]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers*

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

We evaluate our implementations of the package algorithms on Viper's test suite and compare them to Viper's implementation of the FIA as presented in Section 4.2.3. Our key findings are that our algorithms

1. enable the verification of almost all correct `package` operations,
2. correctly report `package` operations that are supposed to fail (in contrast to the FIA), and
3. have an acceptable performance overhead compared to the FIA.

Moreover, interpreting wands as combinable wands as explained in Section 4.4 has only a minor effect on the results, but correctly rejects attempts to package a non-combinable wand. This finding suggests that verifiers could improve their expressiveness by allowing flexible combinations of wands and predicates with only a minor completeness penalty.

For our evaluation, we considered all 85 files in the test suite for Viper's VCG with at least one `package` operation. From these 85 files, we removed 29 files containing features that our implementation does not yet support.[17]

17: Since this evaluation was performed, we have added support for several key missing features, such as support for iterated separating conjunctions on both sides of the wand.

Table 4.1 gives an overview of our results. These confirm that our algorithms for standard and combinable wands (S-Alg and C-Alg) do not produce false negatives, that is, are sound. In contrast, the FIA does verify an incorrect program (which is similar to the example in Section 4.2.3). While this is only a single unsound example, it is worth emphasizing that (a) it comes from the pre-existing test suite of the tool itself, (b) the unsoundness was not known of until our work, and (c) soundness issues in a program verifier are critical to address; we show how to achieve this.

Compared with the FIA, our implementation reports a handful of false positives (spurious errors). For S-Alg, 3 out of 5 false positives are caused

Algorithm	Expected result	Incorrectly verified	Spurious errors
FIA	55	1	0
S-Alg	51	0	5
C-Alg	48	0	8

Table 4.1.: Verification results on our 56 benchmarks with the FIA, our algorithm for standard wands (S-Alg), and for combinable wands (C-Alg). For each algorithm, we report the number of correct verification results, false negatives, and false positives.

by missing features of our implementation (such as remembering a subset of the permissions that are inside predicate instances when manipulating predicates); these features could be straightforwardly added in the future. The other 2 false positives are caused by S-Alg's strategy. In one, the only potential footprint prevents the wand from ever being applied; although technically a false positive, it seems useful to reject the wand and alert the user. The other case is due to a coarse-grained heuristic applied by S-Alg that can be improved.

C-Alg reports the expected result in 48 benchmarks. Importantly, it correctly rejects one wand that indeed does not hold as a combinable wand. 5 of the 8 false positives are identical to those for S-Alg. In the other three benchmarks, the wands still do hold as combinable wands, but further extensions to C-Alg are required to handle them due to technical challenges regarding predicate instances. Once these extensions have been implemented, C-Alg will be as precise as S-Alg, indicating that comparable program verifiers could switch to combinable wands to simply enable sound, flexible combinations with predicates.

To evaluate performance, we ran each of the three implementations 5 times on each of the 56 benchmarks on a Lenovo T480 with 32 GB of RAM and a i7-8550U 1.8 GhZ CPU, running on Windows 10. We removed the slowest and fastest time, and then took the mean of the remaining 3 runs. The FIA takes between 1 and 11 seconds per benchmark. On average, S-Alg is 21% slower than the FIA. For 46 of the 56 examples, the increase is less than 30%, and for 3 examples S-Alg is between a factor 2 and 3.4 slower. The overhead is most likely due to the increased complexity of our algorithms, which track more states explicitly and require more quantified axioms in the Boogie encoding. C-Alg is on average 10% slower than S-Alg. We consider the performance overhead of our algorithms to be acceptable, especially since wands occur much more frequently in our benchmarks than in average Viper projects, as judged by existing tests and examples. More representative projects will, thus, incur a much smaller slow-down.

4.6. Related Work

Before the work based on this chapter was published, VerCors [57] and Viper [16] were, to the best of our knowledge, the only automated SL verifiers that supported magic wands. Both employ `package` and `apply` ghost operations. VerCors' package algorithm requires a user to manually specify a footprint whereas Viper infers footprints using the FIA, which is unsound as we show in Section 4.2.3. Our package algorithm is as automated as the FIA but is sound.

[57]: Blom et al. (2017), *The VerCors Tool Set*

[16]: Müller et al. (2016), *Viper*

Since this work was published, support for magic wands has been added [197] to the Gillian verifier [30] based on the package logic

[197]: Lööw et al. (2024), *Matching Plans for Frame Inference in Compositional Reasoning*

[30]: Fragoso Santos et al. (2020), *Gillian, Part i*

presented here, which has proved important in the context of unsafe Rust verification [64]. The Pulse automated verifier (based on the PulseCore separation logic [50] embedded in F* [14]) supports *trades*, a binary connective stronger than the magic wand, as it allows ghost updates and opening invariants.[18]

[64]: Ayoun et al. (2025), *A Hybrid Approach to Semi-automated Rust Verification*

[50]: Ebner et al. (2025), *PulseCore*

[14]: Swamy et al. (2016), *Dependent Types and Multi-Monadic Effects in F**

18: Similar to a fancy update in Iris [31].

Lee and Park [200] develop a sound and complete proof system for SL including the magic wand. Moreover, they derive a decision procedure from their completeness proof for propositional SL. However, more expressive versions of SL (that include *e.g.*, predicates and quantifiers) are undecidable [196] and so this decision procedure cannot be directly applied in the logics employed by program verifiers.

[200]: Lee et al. (2014), *A Proof System for Separation Logic with Magic Wand*

[196]: Brochenin et al. (2012), *On the Almighty Wand*

Chang *et al.* [201] define a shape analysis that derives magic wands $A \mathrel{-\!\!*} B$ of a restricted form (A and B cannot contain general imprecise assertions); our package logic does not impose such restrictions, which rule out some useful kinds of wands. For example, A may be a data structure with a read-only part expressed via existentially-quantified fractional permissions or A may contain the necessary permission to invoke a method, which may be an arbitrary assertion. In follow-up work, Chang and Rival [189] present a restricted "inductive" magic wand. Footprints of inductive wands are expressed via a finite unrolling of an inductive predicate defining B until the permissions in A are revealed. Such wands are useful to reason about data structures with back-pointers such as doubly-linked lists.

[201]: Chang et al. (2007), *Shape Analysis with Structural Invariant Checkers*

[189]: Chang et al. (2008), *Relational Inductive Shape Analysis*

Iris [31] provides a custom proof mode [202] for interactive SL proofs in Coq [32]. Separation logics expressed in Iris support wands and are more expressive than those of automated SL verifiers at the cost of requiring more user guidance. Packaging a wand in the proof mode requires manually specifying a footprint and proving that the footprint is correct. While tactics can be used in principle to automate parts of this process, there are no specific tactics to infer footprints.

[31]: Jung et al. (2018), *Iris from the Ground Up*

[202]: Krebbers et al. (2018), *MoSeL*

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

Automated Verifiers for Hyperproperties

5. Hyper Hoare Logic

A part of me was shocked and paranoid: Why hasn't anybody else implemented this before?!? This idea seems so obvious in retrospect!

Philip Guo, *The Ph.D Grind*

We now turn to a different but complementary goal: The automated verification of hyperproperties. As explained in Chapter 1, hyperproperties [27] are properties that relate multiple executions of a program, such as determinism (∀∀), transitivity (∀∀∀), generalized non-interference (∀∀∃) [28], or the existence of a minimum (∃∀). This chapter introduces Hyper Hoare Logic, a novel formal foundation for proving program hyperproperties, while the next chapter presents Hypra, a novel automated verifier for hyperproperties based on Hyper Hoare Logic.

[27]: Clarkson et al. (2008), *Hyperproperties*

[28]: McCullough (1988), *Noninterference and the Composability of Security Properties*

5.1. Introduction

As shown by Table 1.2, existing program logics for hyperproperties face two open problems. First, $\exists^*\forall^*$-hyperproperties, such as the existence of a minimum or a violation of generalized non-interference, cannot be expressed by any existing program logic. Second, the existing logics cover different, often disjoint program properties, which may hinder practical applications: reasoning about a wide spectrum of properties of a given program requires the application of several logics, each with its own judgment; properties expressed in different, incompatible logics cannot be composed within the same proof system, and thus within the same automated verifier.

To overcome these limitations, this chapter presents *Hyper Hoare Logic*, a novel program logic that enables *proving or disproving* any *program hyperproperty*, a particular class of hyperproperties over the set of terminating executions of a program (formally defined in Section 5.3.3). In the rest of this chapter, when the context is clear, we use *hyperproperties* to refer to program hyperproperties. Program hyperproperties include many different types of properties, relating *any* (potentially unbounded or even infinite) number of program executions, and many hyperproperties that no existing Hoare logic can handle. Among them are $\exists^*\forall^*$-hyperproperties such as violations of generalized non-interference (Section 5.5.3), and hyperproperties relating an unbounded or infinite number of executions such as quantifying information flow based on Shannon entropy or min-capacity [203–206] (the extended version [207] of this chapter provides an example). Moreover, Hyper Hoare Logic offers rules to compose hyper-triples with different types of hyperproperties, such as monotonicity (∀∀) with the existence of a minimum (∃∀), or non-interference (∀∀) with generalized non-interference (∀∀∃).

Hyper Hoare Logic is based on a simple yet powerful idea: We lift pre- and postconditions from assertions over a *fixed* number of execution

[203]: Shannon (1948), *A Mathematical Theory of Communication*

[204]: Assaf et al. (2017), *Hypercollecting Semantics and Its Application to Static Analysis of Information Flow*

[205]: Yasuoka et al. (2010), *On Bounding Problems of Quantitative Information Flow*

[206]: Smith (2009), *On the Foundations of Quantitative Information Flow*

[207]: Dardinier et al. (2024), *Hyper Hoare Logic*

states (as is usually done by existing Hoare logics for hyperproperties) to *hyper-assertions* over *sets* of execution states. Hyper Hoare Logic then establishes *hyper-triples* of the form $[P]\ C\ [Q]$, where P and Q are hyper-assertions. Such a hyper-triple is valid iff for any set of initial states S that satisfies P, the set of reachable states from executing C in any state from S satisfies Q. By allowing assertions to quantify *universally* over states, Hyper Hoare Logic can express overapproximate properties, whereas *existential* quantification expresses underapproximate properties. Combinations of universal and existential quantification in the same assertion, as well as assertions over infinite sets of states, allow Hyper Hoare Logic to prove or disprove properties beyond existing logics.

Contributions and outline. The main contribution of this chapter is Hyper Hoare Logic, a novel Hoare logic that can prove or disprove arbitrary hyperproperties over terminating executions. More precisely, our contributions are:

- We formalize a novel notion of *hyper-triples*, and demonstrate their expressiveness on judgments of existing Hoare logics and on hyperproperties that no existing Hoare logic supports. We prove that hyper-triples capture precisely program hyperproperties, and that any invalid hyper-triple can be disproved by proving another hyper-triple. (Section 5.3)
- We present a minimal set of core rules for Hyper Hoare Logic, and prove that these rules are sound and complete for establishing valid hyper-triples. (Section 5.4)
- We formally define a notion of *syntactic hyper-assertions*, which restricts the interaction with the set of states to universal and existential quantification over states, and derive easy-to-use syntactic rules for these syntactic hyper-assertions. (Section 5.5)
- We present novel loop rules, which capture important reasoning principles. (Section 5.6)
- We present compositionality rules for hyper-triples, which enable the flexible composition of hyper-triples of different forms and, thus, facilitate modular proofs. (Section 5.7)

Moreover, Section 5.2 presents hyper-triples informally, and shows how they can be used to specify hyperproperties, and Section 5.8 discusses related work.

All the technical results presented in this chapter (including the soundness of all the rules we present) have been proved in Isabelle/HOL [33], and our mechanization is publicly available [135, 136].

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

[135]: Dardinier (2023), *Formalization of Hyper Hoare Logic: A Logic to (Dis-)Prove Program Hyperproperties*

[136]: Dardinier et al. (2024), *Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties (Artifact)*

5.2. Hyper-Triples, Informally

In this section, we illustrate how hyper-triples can be used to express different types of hyperproperties, including over- and underapproximate hyperproperties for single (Section 5.2.1) and multiple (Section 5.2.2 and Section 5.2.3) executions.

5.2.1. Overapproximation and Underapproximation

Example 5.2.1 A random number generator.
Consider the command $C_0 \triangleq (x := \mathit{randIntBounded}(0, 9))$, which generates a random integer between 0 and 9 (both included), and assigns it to the variable x. We want C_0 to satisfy the following two functional correctness properties:

(P1) The final value of x is in the interval $[0, 9]$.
(P2) Every value in $[0, 9]$ can occur for every initial state (*i.e.*, the output is not determined by the initial state).

Property P1 expresses the *absence* of *bad* executions, in which the output x falls outside the interval $[0, 9]$. This property can be expressed in classical Hoare logic, with the triple $\{\top\}\ C_0\ \{0 \leq x \leq 9\}$. In Hyper Hoare Logic, where assertions are properties of sets of states, we express it using a postcondition that *universally* quantifies over all possible final states: In all final states, the value of x should be in $[0, 9]$. The hyper-triple

$$[\top]\ C_0\ [\forall \langle \varphi' \rangle .\, 0 \leq \varphi'(x) \leq 9]$$

expresses this property. The postcondition, written in the syntax that will be introduced in Section 5.5, is semantically equivalent to $\lambda S'.\, \forall \varphi' \in S'.\, 0 \leq \varphi'(x) \leq 9$. This hyper-triple means that, for any set S of initial states φ (satisfying the trivial precondition $\top$), the set S' of all final states φ' that can be reached by executing C_0 in some initial state $\varphi \in S$ satisfies the postcondition, *i.e.*, all final states $\varphi' \in S'$ have a value for x between 0 and 9. This hyper-triple illustrates a systematic way of expressing classical Hoare triples as hyper-triples (see Appendix A.3.1).

In contrast, property P2 expresses the *existence* of *good* executions and can be expressed using an underapproximate Hoare logic (such as *Incorrectness Logic* [89]). In Hyper Hoare Logic, we use a postcondition that *existentially* quantifies over all possible final states: For each $n \in [0, 9]$, there exists a final state where $x = n$. The hyper-triple

$$[\exists \langle \varphi \rangle .\, \top]\ C_0\ [\forall n .\, 0 \leq n \leq 9 \Rightarrow \exists \langle \varphi' \rangle .\, \varphi'(x) = n]$$

expresses P2. The precondition is semantically equivalent to $\lambda S.\, \exists \varphi \in S$, and the postcondition to $\lambda S'.\, \forall n .\, 0 \leq n \leq 9 \Rightarrow \exists \varphi' \in S'.\, \varphi'(x) = n$. It requires the set S of initial states to be non-empty (otherwise the set of states reachable from states in S by executing C_0 would also be empty, and the postcondition would not hold). The postcondition ensures that, for any $n \in [0, 9]$, it is possible to reach at least one state φ' with $\varphi'(x) = n$.

This example shows that hyper-triples can express both underapproximate and overapproximate properties, which allows Hyper Hoare Logic to reason about both the *absence* of bad executions and the *existence* of good executions. Moreover, hyper-triples can also be used to prove the existence of *incorrect* executions, which has proven useful in practice to find bugs without false positives [89, 101]. To the best of our knowledge, the only other Hoare logics that can express both properties P1 and P2 are Outcome Logic [102], Exact Separation Logic [120], and BiKAT [105].[1] However, the first two are limited to properties of individual executions,

[89]: O'Hearn (2019), *Incorrectness Logic*

[89]: O'Hearn (2019), *Incorrectness Logic*
[101]: Le et al. (2022), *Finding Real Bugs in Big Programs with Incorrectness Logic*
[102]: Zilberstein et al. (2023), *Outcome Logic*
[120]: Maksimović et al. (2023), *Exact Separation Logic*
[105]: Antonopoulos et al. (2023), *An Algebra of Alignment for Relational Verification*

1: While RHLE [96] can in principle reason about the existence of executions, it is unclear how to express the existence *for all* numbers n.

and the latter to properties relating two executions. Thus, these logics cannot handle the general class of k-safety hyperproperties, which we discuss next.

5.2.2. (Dis-)Proving k-Safety Hyperproperties

A *k-safety hyperproperty* [27] is a property that characterizes *all combinations of k* executions of the same program. An important example (for $k = 2$) is information flow security [29] which requires that programs that manipulate secret data (such as passwords) do not expose secret information to their users. In other words, the content of high-sensitivity (secret) variables must not leak into low-sensitivity (public) variables. For deterministic programs, information flow security is often formalized as *non-interference* (NI) [208], a 2-safety hyperproperty: Any two executions of the program with the same low-sensitivity (*low* for short) inputs (but potentially different high-sensitivity inputs) must have the same low outputs. That is, for all pairs of executions τ_1 and τ_2, if τ_1 and τ_2 agree on the initial values of all low variables, they must also agree on the final values of all low variables. This ensures that the final values of low variables are not influenced by the values of high variables.

[27]: Clarkson et al. (2008), *Hyperproperties*

[29]: Goguen et al. (1982), *Security Policies and Security Models*

[208]: Volpano et al. (1996), *A Sound Type System for Secure Flow Analysis*

Example 5.2.2 Expressing non-interference in Hyper Hoare Logic**.** Assuming that we have one low input variable l, one high input variable h, and one low output variable o, the hyper-triple

$$\underbrace{[\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \varphi_1(l) = \varphi_2(l)]}_{low(l)}\ o \coloneqq f(l,h)\ \underbrace{[\forall\langle\varphi_1'\rangle,\langle\varphi_2'\rangle.\ \varphi_1'(o) = \varphi_2'(o)]}_{low(o)}$$

expresses that $C_1 \triangleq (o \coloneqq f(l,h))$ satisfies NI: If all states in S have the same value for l, then all final states reachable by executing C_1 in any initial state $\varphi \in S$ will have the same value for o. This set-based definition is equivalent to the standard definition based on pairs of executions, and instantiating S with a set of two states directly yields the standard definition.

Non-interference requires that all final states have the same value for l, which is a *symmetric* property. Other k-safety hyperproperties, such as monotonicity, are asymmetric, and thus need to relate initial and final states.

Example 5.2.3 Expressing monotonicity in Hyper Hoare Logic**.** The program $y \coloneqq f(x)$ is *monotonic* iff for any two executions with $\varphi_1(x) \geq \varphi_2(x)$, we have $\varphi_1'(y) \geq \varphi_2'(y)$, where φ_1 and φ_2 are the initial states φ_1' and φ_2' are the *corresponding* final states. To relate initial and final states, Hyper Hoare Logic uses *logical variables* (also called *auxiliary variables* [209]). These variables cannot appear in a program, and thus are guaranteed to have the same values in the initial and final states of an execution. We use this property to tag corresponding

[209]: Kleymann (1999), *Hoare Logic and Auxiliary Variables*

states, as illustrated by the following hyper-triple for monotonicity:

$$[\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \varphi_1(t) = 1 \land \varphi_2(t) = 2 \Rightarrow \varphi_1(x) \geq \varphi_2(x)]$$
$$y \coloneqq f(x)$$
$$[\forall\langle\varphi'_1\rangle, \langle\varphi'_2\rangle.\ \varphi'_1(t) = 1 \land \varphi'_2(t) = 2 \Rightarrow \varphi'_1(y) \geq \varphi'_2(y)]$$

Here, t is a logical variable used to distinguish the two executions of the program.

Disproving k-safety hyperproperties. As explained in Chapter 1, being able to prove that a property does *not* hold is valuable in practice, because it allows building tools that can find bugs without false positives. Hyper Hoare Logic is able to *disprove* hyperproperties by proving a hyperproperty that is essentially its negation.

Example 5.2.4 Expressing a violation of non-interference.
We can express that the insecure program $C_2 \triangleq$ (**if** $(h > 0)$ $\{o \coloneqq l\}$ **else** $\{o \coloneqq 5\}$), where h is a high variable containing a confidential value, *violates* non-interference (NI), using the following hyper-triple:

$$[\mathit{low}(l) \land (\exists\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \varphi_1(h) > 0 \land \varphi_2(h) \leq 0)]$$
$$\textbf{if } (h > 0)\ \{o \coloneqq l\}\ \textbf{else}\ \{o \coloneqq 5\}$$
$$[\exists\langle\varphi'_1\rangle, \langle\varphi'_2\rangle.\ \varphi'_1(o) \neq \varphi'_2(o)]$$

The postcondition is the negation of the postcondition for non-interference above (Example 5.2.2), hence expressing that C_2 *violates* NI. Note that the precondition needs to be stronger than in Example 5.2.2: Since the postcondition has to hold for *all* sets that satisfy the precondition, we must require the set of initial states to contain at least two states that will definitely lead to different final values of o.

The general class of k-safety hyperproperties includes properties that relate more than 2 executions, such as transitivity ($k = 3$) and associativity ($k = 4$) [88]. The only other Hoare logic that can be used to both prove and disprove arbitrary k-safety hyperproperties is RHLE [96], since it supports $\forall^*\exists^*$-hyperproperties, which includes both safety hyperproperties (*i.e.*, $\forall^*$-hyperproperties) and their negation (*i.e.*, $\exists^*$-hyperproperties). However, RHLE does not support $\exists^*\forall^*$-hyperproperties, and thus cannot disprove $\forall^*\exists^*$-hyperproperties such as generalized non-interference, as we discuss next.

[88]: Sousa et al. (2016), *Cartesian Hoare Logic for Verifying K-Safety Properties*

[96]: Dickerson et al. (2022), *RHLE*

5.2.3. Beyond k-Safety

NI is widely used to express information flow security for deterministic programs, but is overly restrictive for non-deterministic programs.

Example 5.2.5 A secure, non-deterministic program that violates non-interference.
The program $C_3 \triangleq (y \coloneqq \mathit{nonDet}();\ o \coloneqq h + y)$ is information flow se-

cure. Since the secret h is added to an unbounded non-deterministically chosen integer y, any secret h can result in any[2] value for the public variable o and, thus, we cannot learn anything certain about h from observing the value of o. However, because of non-determinism, C_3 does not satisfy NI: Two executions with the same initial values for l could get different values for y, and thus have different final values for o.

2: This property holds for both unbounded and bounded arithmetic.

Information flow security for non-deterministic programs (such as C_3) is often formalized as *generalized non-interference* (GNI) [210, 211] (also called *possibilistic non-interference*), a security notion weaker than NI. GNI allows two executions τ_1 and τ_2 with the same low inputs to have *different* low outputs, provided that there is a third execution τ with the same low inputs that has the same high inputs as τ_1 and the same low outputs as τ_2. That is, the difference in the low outputs between τ_1 and τ_2 cannot be attributed to their secret inputs.[3]

[210]: McCullough (1987), *Specifications for Multi-Level Security and a Hook-Up*
[211]: McLean (1996), *A General Theory of Composition for a Class of "Possibilistic" Properties*

Example 5.2.6 Expressing generalized non-interference.
The non-deterministic program $C_3 \triangleq (y \coloneqq \mathit{nonDet}();\ o \coloneqq h + y)$ satisfies GNI, which can be expressed via the following hyper-triple:

$$[\mathit{low}(l)]\ C_3\ [\forall\langle\varphi_1'\rangle, \langle\varphi_2'\rangle.\ \exists\langle\varphi'\rangle.\ \varphi'(h) = \varphi_1'(h) \land \varphi'(o) = \varphi_2'(o)]$$

The final states φ_1' and φ_2' correspond to the executions τ_1 and τ_2, respectively, and φ' corresponds to execution τ.

3: GNI is often formulated without the requirement that τ_1 and τ_2 have the same low inputs, *e.g.*, in Clarkson and Schneider [27]. This alternative formulation can also be expressed in Hyper Hoare Logic, with the precondition $\forall\langle\varphi\rangle.\ \varphi(l_{in}) = \varphi(l)$ and the postcondition $\forall\langle\varphi_1'\rangle, \langle\varphi_2'\rangle.\ \exists\langle\varphi'\rangle.\ \varphi'(h) = \varphi_1'(h) \land \varphi'(l_{in}) = \varphi_2'(l_{in}) \land \varphi'(o) = \varphi_2'(o)$. The precondition binds, in each state, the initial value of l to the logical variable l_{in}, which enables the postcondition to refer to the initial value of l.

As before, the expressivenes of hyper-triples enables us not only to express that a program *satisfies* complex hyperproperties such as GNI, but also that a program *violates* them.

Example 5.2.7 Expressing a violation of generalized non-interference.
The program $C_4 \triangleq (y \coloneqq \mathit{nonDet}();\ \textbf{assume}\ y \leq 9;\ o \coloneqq h + y)$, where the first two statements model a non-deterministic choice of y smaller or equal to 9, leaks information: Observing for example $o = 20$ at the end of an execution, one can deduce that $h \geq 11$ (because $y \leq 9$). We can formally express that C_4 violates GNI with the following hyper-triple:[4]

$$[\mathit{low}(l) \land (\exists\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \varphi_1(h) \neq \varphi_2(h))]$$
$$y \coloneqq \mathit{nonDet}();\ \textbf{assume}\ y \leq 9;\ o \coloneqq h + y$$
$$[\exists\langle\varphi_1'\rangle, \langle\varphi_2'\rangle.\ \forall\langle\varphi'\rangle.\ \varphi'(h) = \varphi_1'(h) \Rightarrow \varphi'(o) \neq \varphi_2'(o)]$$

The postcondition implies the negation of the postcondition we used previously to express GNI. As before, we had to strengthen the precondition to prove this violation.

4: Still assuming that h is not modified.

GNI is a $\forall\forall\exists$-hyperproperty, whereas its negation is an $\exists\exists\forall$-hyperproperty. To the best of our knowledge, Hyper Hoare Logic is the only Hoare logic that can both prove and disprove GNI. In fact, we will see in Section 5.3.3 that all hyperproperties over terminating program executions can be proven or disproven with Hyper Hoare Logic.

$$\langle \textbf{skip}, \sigma\rangle \to \sigma \qquad \langle x := e, \sigma\rangle \to \sigma[x \mapsto e(\sigma)] \qquad \langle x := \mathit{nonDet}(), \sigma\rangle \to \sigma[x \mapsto v] \qquad \frac{\langle C_1, \sigma\rangle \to \sigma' \quad \langle C_2, \sigma'\rangle \to \sigma''}{\langle C_1; C_2, \sigma\rangle \to \sigma''}$$

$$\frac{\langle C_1, \sigma\rangle \to \sigma'}{\langle C_1 + C_2, \sigma\rangle \to \sigma'} \qquad \frac{\langle C_2, \sigma\rangle \to \sigma'}{\langle C_1 + C_2, \sigma\rangle \to \sigma'} \qquad \frac{b(\sigma)}{\langle \textbf{assume } b, \sigma\rangle \to \sigma} \qquad \frac{\langle C, \sigma\rangle \to \sigma' \quad \langle C^*, \sigma'\rangle \to \sigma''}{\langle C^*, \sigma\rangle \to \sigma''} \qquad \langle C^*, \sigma\rangle \to \sigma$$

Figure 5.1.: Big-step semantics. Since expressions are functions from states to values, $e(\sigma)$ denotes the evaluation of expression e in state σ. $\sigma[x \mapsto v]$ is the state that yields v for x and the value in σ for all other variables.

5.3. Hyper-Triples, Formally

In this section, we define the programming language used in this chapter (Section 5.3.1), formalize hyper-triples (Section 5.3.2), and formally characterize the expressiveness of hyper-triples (Section 5.3.3). All technical results presented in this section have been formalized in Isabelle/HOL.

5.3.1. Language and Semantics

We present Hyper Hoare Logic for the following imperative programming language:

Definition 5.3.1 Program states and programming language**.**
*A **program state** (ranged over by σ) is a mapping from local variables (in the set PVars) to values (in the set PVals): The set of program states PStates is defined as the set of total functions from PVars to PVals: PStates $\triangleq$ PVars $\to$ PVals.*

Program commands C are defined by the following syntax, where x ranges over variables in the set PVars, e over expressions (modeled as total functions from PStates to PVals), and b over predicates over states (total functions from PStates to Booleans):

$$C ::= \textbf{skip} \mid x := e \mid x := \mathit{nonDet}() \mid \textbf{assume } b \mid C; C \mid C + C \mid C^*$$

The **skip**, assignment, and sequential composition commands are standard. The command **assume** b acts like **skip** if b holds and otherwise stops the execution. Instead of including *deterministic* if-statements and while loops, we consider a *non-deterministic* choice $C_1 + C_2$ and a *non-deterministic* iteration C^*, which are more expressive. Combined with the **assume** command, they can express deterministic if-statements and while loops as follows:

$$\begin{aligned} \textbf{if } (b)\ \{C_1\}\ \textbf{else}\ \{C_2\} &\triangleq (\textbf{assume } b; C_1) + (\textbf{assume } \neg b; C_2) \\ \textbf{if } (b)\ \{C\} &\triangleq (\textbf{assume } b; C) + (\textbf{assume } \neg b) \\ \textbf{while } (b)\ \{C\} &\triangleq (\textbf{assume } b; C)^*; \textbf{assume } \neg b \end{aligned}$$

Our language also includes a non-deterministic assignment $y := \mathit{nonDet}()$ (also called *havoc*), which allows us to model unbounded non-determinism. Together with **assume**, it can for instance model the generation of random numbers between bounds: $y := \mathit{randIntBounded}(a, b)$ can be modeled as $y := \mathit{nonDet}()$; **assume** $a \leq y \leq b$.

The big-step semantics of our language is standard, and formally defined in Figure 5.1. The rule for $x := nonDet()$ allows x to be updated with any value v. **assume** b leaves the state unchanged if b holds; otherwise, the semantics gets stuck to indicate that there is no execution in which b does *not* hold. The command $C_1 + C_2$ non-deterministically executes either C_1 or C_2. C^* non-deterministically either performs another loop iteration or terminates.

Note that our language does not contain any command that could fail (in particular, expression evaluation is total, such that division-by-zero and other errors cannot occur). Runtime failures could easily be modeled by instrumenting the program with a special Boolean variable *err* that tracks whether a runtime error has occurred and skips the rest of the execution if this is the case.

5.3.2. Hyper-Triples

As explained in Section 5.2, the key idea behind Hyper Hoare Logic is to use *predicates over sets of states* as pre- and postconditions, whereas traditional Hoare logics use properties of individual states (or of a given number k of states in logics for hyperproperties). Considering arbitrary sets of states increases the expressiveness of triples substantially; for instance, universal and existential quantification over these sets corresponds to over- and underapproximate reasoning, respectively. Moreover, combining both forms of quantification allows one to express advanced hyperproperties, such as generalized non-interference (Example 5.2.6).

To allow the assertions of Hyper Hoare Logic to refer to logical variables (as motivated in Section 5.2.2), we include them in our notion of state.

Definition 5.3.2 Extended states**.**
An ***extended state*** *(ranged over by φ) is a pair of a* logical state *(a total mapping from logical variables to logical values) and a program state:*

$$ExtStates \triangleq (LVars \to LVals) \times PStates$$

Given an extended state φ, we write φ^L to refer to the logical state and φ^P to refer to the program state, that is, $\varphi = (\varphi^L, \varphi^P)$.

We use the same meta variables (x, y, z) for program and logical variables. When it is clear from the context that $x \in PVars$ (resp. $x \in LVars$), we often write $\varphi(x)$ to denote $\varphi^P(x)$ (resp. $\varphi^L(x)$).

The assertions of Hyper Hoare Logic are predicates over sets of extended states:

Definition 5.3.3 Hyper-assertions**.**
A ***hyper-assertion*** *(ranged over by P, Q, R) is a total function from $\mathbb{P}(ExtStates)$ to Booleans.*

A hyper-assertion P ***entails*** *a hyper-assertion Q, written $P \models Q$, iff all sets that satisfy P also satisfy Q:*

$$(P \models Q) \triangleq (\forall S.\, P(S) \Rightarrow Q(S))$$

We formalize hyper-assertions as semantic properties, which allows us to focus on the key ideas of the logic. In Section 5.5, we will define a syntax for hyper-assertions, which will allow us to derive simpler rules than the core rules presented in the next section.

To formalize the meaning of hyper-triples, we need to relate them formally to the semantics of our programming language. Since hyper-triples are defined over extended states, we first define a semantic function *sem* that lifts the operational semantics to extended states; it yields the set of extended states that can be reached by executing a command C from a set of extended states S:

Definition 5.3.4 Extended semantics**.**

$$sem(C, S) \triangleq \{\varphi \mid \exists \sigma.\, (\varphi^L, \sigma) \in S \wedge \langle C, \sigma\rangle \to \varphi^P\}$$

Lemma 5.3.1 Properties of the extended semantics**.**
The extended semantics satisfies the following useful properties:

1. $sem(C, S_1 \cup S_2) = sem(C, S_1) \cup sem(C, S_2)$
2. $S \subseteq S' \Rightarrow sem(C, S) \subseteq sem(C, S')$
3. $sem(C, \bigcup_x f(x)) = \bigcup_x sem(C, f(x))$
4. $sem(\textbf{skip}, S) = S$
5. $sem(C_1; C_2, S) = sem(C_2, sem(C_1, S))$
6. $sem(C_1 + C_2, S) = sem(C_1, S) \cup sem(C_2, S)$
7. $sem(C^*, S) = \bigcup_{n \in \mathbb{N}} sem(C^n, S)$ *where* $C^n \triangleq \underbrace{C;\ \ldots;\ C}_{n \text{ times}}$

Using the extended semantics, we can now define the meaning of hyper-triples.

Definition 5.3.5 Hyper-triples**.**
Given two hyper-assertions P and Q, and a command C, the ***hyper-triple*** $[P]\ C\ [Q]$ *is* ***valid****, written* $\models [P]\ C\ [Q]$*, iff for any set S of initial extended states that satisfies P, the set $sem(C, S)$ of extended states reachable by executing C in some state from S satisfies Q:*

$$\models [P]\ C\ [Q] \triangleq (\forall S.\, P(S) \Rightarrow Q(sem(C, S)))$$

This definition is similar to classical Hoare logic, where the initial and final states have been replaced by *sets* of extended states. As we have seen in Section 5.2, hyper-assertions over sets of states allow our hyper-triples to express properties of single executions and of multiple executions (*i.e.*, hyperproperties), as well as to perform overapproximate reasoning (like *e.g.*, Hoare Logic) and underapproximate reasoning (like *e.g.*, Incorrectness Logic).

Terminating hyper-triples

Definition 5.3.5 makes Hyper Hoare Logic a "partial correctness" logic, in the sense that non-terminating executions are ignored by hyper-triples. Alternatively, we can define a stronger notion of *terminating*

hyper-triples, written $\models_{\Downarrow} [P]\ C\ [Q]$, as follows:

$$\begin{aligned} &\models_{\Downarrow} [P]\ C\ [Q] \\ &\triangleq \big(\forall S.\, P(S) \Rightarrow (Q(\mathit{sem}(C,S)) \wedge (\forall \varphi \in S.\, \exists \sigma'.\, \langle C, \varphi^P\rangle \rightarrow \sigma'))\big) \end{aligned}$$

The first conjunct corresponds to the validity of hyper-triples from Definition 5.3.5, while the second conjunct additionally ensures the existence of (at least) one terminating execution from any initial state from S.

Note that, for non-deterministic programs, terminating hyper-triples ensure a property *strictly weaker* than *total correctness* (*i.e.*, the absence of non-terminating executions). For example, the terminating hyper-triple

$$\vdash_{\Downarrow} [\top]\ x := \mathit{nonDet}();\ \textbf{while}\ (x > 0)\ \{\textbf{skip}\}\ [\top]$$

is valid according to this definition, even though (infinitely) many executions do not terminate.

To formally characterize total correctness, we would need to change our semantic model, as our big-step semantics (Figure 5.1) ignores non-terminating executions. One possibility would be to use a small-step semantics instead, and adapt Definition 5.3.4 accordingly. However, as we will see later (in Section 5.7.1 and Section 5.6.1), this weaker property is sufficient to obtain inference rules that lift restrictions compared to their non-terminating hyper-triple counterparts.

5.3.3. Expressiveness of Hyper-Triples

We now show that hyper-triples are expressive enough to capture arbitrary hyperproperties over finite program executions. A *hyperproperty* [27] is traditionally defined as a property of sets of *traces* of a system, that is, of sequences of system states. Since Hoare logics typically consider only the initial and final state of a program execution, we use a slightly adapted definition here:

[27]: Clarkson et al. (2008), *Hyperproperties*

Definition 5.3.6 Program hyperproperties**.**
A ***program hyperproperty*** *is a set of sets of pairs of program states,* i.e.*, an element of* $\mathbb{P}(\mathbb{P}(\mathit{PStates} \times \mathit{PStates}))$.

A command C *satisfies the program hyperproperty* $\mathcal{H}$ *iff the set of all pairs of pre- and post-states of* C *is an element of* $\mathcal{H}$*:* $\{(\sigma, \sigma') \mid \langle C, \sigma\rangle \rightarrow \sigma'\} \in \mathcal{H}$.

Equivalently, a program hyperproperty can be thought of as a predicate over $\mathbb{P}(\mathit{PStates} \times \mathit{PStates})$. Note that this definition subsumes properties of single executions, such as functional correctness properties.

In contrast to traditional hyperproperties, our program hyperproperties describe only the *finite* executions of a program, that is, those that reach a final state. An extension of Hyper Hoare Logic to infinite executions might be possible by defining hyper-assertions over sets of traces rather than sets of states; we leave this as future work. In the rest of this chapter, when the context is clear, we use *hyperproperties* to refer to *program hyperproperties*.

Any program hyperproperty can be expressed as a hyper-triple in Hyper Hoare Logic:

Theorem 5.3.2 Expressing hyperproperties as hyper-triples**.**
Let $\mathcal{H}$ be a program hyperproperty. Assume that the cardinality of LVars is at least the cardinality of PVars, and that the cardinality of LVals is at least the cardinality of PVals.

Then there exist hyper-assertions P and Q such that, for any command C, $C \in \mathcal{H}$ iff $\models [P]\ C\ [Q]$.

Proof. We define the precondition P such that the set of initial states contains all program states, and the values of all program variables in these states are recorded in logical variables (which is possible due to the cardinality assumptions). Since the logical variables are not affected by the execution of C, they allow Q to refer to the initial values of any program variable, in addition to their values in the final state. Consequently, Q can describe all possible pairs of pre- and post-states. We simply define Q to be true iff the set of these pairs is contained in $\mathcal{H}$. □

We also proved the converse: every hyper-triple describes a program hyperproperty. That is, hyper-triples capture exactly the hyperproperties over finite executions.

Theorem 5.3.3 Expressing hyper-triples as hyperproperties**.**
For any hyper-assertions P and Q, there exists a hyperproperty $\mathcal{H}$ such that, for any command C, $C \in \mathcal{H}$ iff $\models [P]\ C\ [Q]$.

Proof. We define

$$\mathcal{H} \triangleq \{\Sigma \mid \forall S.\, P(S) \Rightarrow Q(\{(l, \sigma') \mid \exists \sigma.\, (l, \sigma) \in S \wedge (\sigma, \sigma') \in \Sigma\})\}$$

□

Combined with the fact that our logic is complete, as we will in the next section (Theorem 5.4.2), this theorem implies that, if a command C satisfies a hyperproperty $\mathcal{H}$ then there exists a proof of it in Hyper Hoare Logic. More surprisingly, our logic also allows us to *disprove* any hyperproperty: If C does *not* satisfy $\mathcal{H}$ then C satisfies the *complement* of $\mathcal{H}$, which is also a hyperproperty, and thus can also be proved. Consequently, Hyper Hoare Logic can prove or disprove any *program hyperproperty* as defined in Definition 5.3.6.

Since hyper-triples express hyperproperties (Theorem 5.3.2 and Theorem 5.3.3), the ability to disprove any hyperproperty implies that Hyper Hoare Logic can also disprove any *hyper-triple*. More precisely, one can *always* use Hyper Hoare Logic to prove that some hyper-triple $[P]\ C\ [Q]$ is *invalid*, by proving the validity of another hyper-triple $[P']\ C\ [\neg Q]$, where P' is a satisfiable hyper-assertion that entails P. Conversely, the validity of such a hyper-triple $[P']\ C\ [\neg Q]$ implies that all hyper-triples $[P]\ C\ [Q]$ (with P weaker than P') are *invalid*. The following theorem precisely expresses this observation:

Theorem 5.3.4 Disproving hyper-triples.
Given P, C, and Q, the following two propositions are equivalent:

(1) $\models [P]\ C\ [Q]$ *does not hold.*
(2) There exists a hyper-assertion P' that is satisfiable, entails P, and $\models [P']\ C\ [\neg Q]$.

Proof. By negating Definition 5.3.5, we get that point (1) is equivalent to the existence of a set of extended states S such that $P(S)$ holds but $Q(sem(C, S))$ does not, *i.e.*, $\neg Q(sem(C, S))$ holds. Let $P' \triangleq (\lambda S'.\ S = S')$. P' is clearly satisfiable. Moreover, point (1) implies that P' entails P, and that $\models [P']\ C\ [\neg Q]$ holds. Thus, (1) implies (2).

Assuming (2), we get that there exists a set of extended states S such that $P'(S)$ (since P' is satisfiable) and $\neg Q(sem(C, S))$ hold. Since P' entails P, $P(S)$ holds, which implies (1). □

We need to strengthen P to P' in point (2), because there might be some sets S, S' that both satisfy P, such that $Q(sem(C, S))$ holds, but $Q(sem(C, S'))$ does not. This was the case for Example 5.2.4 and Example 5.2.7 in Section 5.2.2 and Section 5.2.3; for instance, the precondition from Example 5.2.7 was strengthened to include $\exists \langle \varphi_1 \rangle, \langle \varphi_2 \rangle.\ \varphi_1(h) \neq \varphi_2(h)$.

Theorem 5.3.4 is another illustration of the expressiveness of Hyper Hoare Logic. The corresponding result does *not* hold in traditional Hoare logics:

Example 5.3.1 Hoare logic is not expressive enough to disprove its invalid triples.
The classical Hoare triple $\{\top\}\ x := nonDet()\ \{x \geq 5\}$ does not hold (1), but there is *no* satisfiable P such that $\{P\}\ x := nonDet()\ \{\neg(x \geq 5)\}$ holds (2). Indeed, to disprove this triple, one needs to show the *existence* of an execution that satisfies the negated postcondition, which is not expressible in standard Hoare logic. In contrast, Hyper Hoare Logic can disprove the classical Hoare triple by proving the hyper-triple

$$[\exists \langle \varphi \rangle.\ \top]\ x := nonDet()\ [\neg(\forall \langle \varphi \rangle.\ \varphi(x) \geq 5)]$$

Disproving termination

As Hyper Hoare Logic can be used to prove and disprove hyperproperties, we could extend Hyper Hoare Logic to prove and disprove termination. Termination can be *proven* with *terminating hyper-triples*, as discussed in Section 5.3.2. Termination can be *disproven* as follows.

To prove *non*-termination of a loop **while** (b) $\{C\}$, one can express and prove that a set of states R, in which all states satisfy the loop guard b, is a *recurrent set* [212]. R is a recurrent set iff executing C in any state from R leads to at least another state in R, which can easily be expressed as a hyper-triple:

$$[\exists \langle \varphi \rangle.\ \varphi \in R]\ \textbf{assume}\ b; C\ [\exists \langle \varphi \rangle.\ \varphi \in R]$$

[212]: Gupta et al. (2008), *Proving Non-Termination*

Thus, if one state from R reaches **while** (b) $\{C\}$, we know that there is at least one non-terminating execution. This idea is for example used by Raad et al. [213], who extend an underapproximate program logic to prove non-termination.

Note that, to formally characterize non-termination, as to formally characterize total correctness, we would need to change our semantic model (for example by using a small-step semantics instead of a big-step semantics).

[213]: Raad et al. (2024), *Non-Termination Proving at Scale*

The correspondence between hyper-triples and program hyperproperties (Theorem 5.3.2 and Theorem 5.3.3), together with the completeness result (Theorem 5.4.2, presented in the next section) precisely characterizes the expressiveness of Hyper Hoare Logic. In Appendix A.3, we show systematic ways to express the judgments of existing over- and underapproximating Hoare logics as hyper-triples.

For example, for P and Q being sets of (extended) states, we formally show that the Hoare logic [8, 9] triple $\models_{HL} \{P\}\ C\ \{Q\}$ is equivalent to the hyper-triple $[\forall\langle\varphi\rangle.\ \varphi \in P]\ C\ [\forall\langle\varphi\rangle.\ \varphi \in Q]$ (Proposition A.3.2), the sufficient incorrectness logic [214] triple $\models_{SIL} \{P\}\ C\ \{Q\}$ is equivalent to the hyper-triple $[\exists\langle\varphi\rangle.\ \varphi \in P]\ C\ [\exists\langle\varphi\rangle.\ \varphi \in Q]$ (Proposition A.3.9), and the incorrectness logic [89] triple $\models_{IL} \{P\}\ C\ \{Q\}$ is equivalent to the hyper-triple $[\lambda S.\ P \subseteq S]\ C\ [\lambda S.\ Q \subseteq S]$ (Proposition A.3.6). As another example, for P and Q being sets of *pairs of* (extended) states, we show (Proposition A.3.4) that the relational Hoare logic [103] triple $\models_{RHL} \{P\}\ C\ \{Q\}$ (as a special case of Cartesian Hoare logic triples [88]) is equivalent to the hyper-triple

[8]: Floyd (1967), *Assigning Meanings to Programs*
[9]: Hoare (1969), *An Axiomatic Basis for Computer Programming*
[214]: Ascari et al. (2024), *Sufficient Incorrectness Logic*
[89]: O'Hearn (2019), *Incorrectness Logic*
[103]: Benton (2004), *Simple Relational Correctness Proofs for Static Analyses and Program Transformations*
[88]: Sousa et al. (2016), *Cartesian Hoare Logic for Verifying K-Safety Properties*

$$[\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \varphi_1(t) = 1 \Rightarrow \varphi_2(t) = 2 \Rightarrow (\varphi_1, \varphi_2) \in P]$$
$$C$$
$$[\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \varphi_1(t) = 1 \Rightarrow \varphi_2(t) = 2 \Rightarrow (\varphi_1, \varphi_2) \in Q]$$

5.4. Core Rules

We now present, in Section 5.4.1, a minimal set of *core rules* for compositionally establishing hyper-triples. As we will prove in Section 5.4.2, these core rules are *sound* and *complete*, *i.e.*, every triple that can be derived using these core rules is valid (according to Definition 5.3.5), and every valid triple can be derived with these core rules. All technical results presented in this section have been formalized in Isabelle/HOL.

5.4.1. The Rules

The core rules of Hyper Hoare Logic are shown in Figure 5.2. The rules Skip, Seq, Cons, and Exist are analogous to traditional Hoare logic. Assume, Assign, and Havoc are straightforward given the semantics of these commands. All three rules work backward. In particular, the precondition of Assume applies the postcondition P only to those states that satisfy the assumption b. By leaving the value v unconstrained, Havoc considers as precondition the postcondition P for all possible values for x. The three rules Assume, Assign, and Havoc are optimized for expressivity; we

$$\textsc{Seq}\ \frac{\vdash [P]\ C_1\ [R] \qquad \vdash [R]\ C_2\ [Q]}{\vdash [P]\ C_1; C_2\ [Q]} \qquad \textsc{Choice}\ \frac{\vdash [P]\ C_1\ [Q_1] \qquad \vdash [P]\ C_2\ [Q_2]}{\vdash [P]\ C_1 + C_2\ [Q_1 \otimes Q_2]}$$

$$\textsc{Iter}\ \frac{\vdash [I_n]\ C\ [I_{n+1}]}{\vdash [I_0]\ C^*\ [\otimes_{n\in\mathbb{N}}\ I_n]} \qquad \textsc{Cons}\ \frac{P \models P' \qquad Q' \models Q \qquad \vdash [P']\ C\ [Q']}{\vdash [P]\ C\ [Q]} \qquad \textsc{Exist}\ \frac{\forall x.\, (\vdash [P_x]\ C\ [Q_x])}{\vdash [\exists x.\, P_x]\ C\ [\exists x.\, Q_x]}$$

$$\textsc{Havoc}\ \vdash [\lambda S.\, P(\{\varphi \mid \exists \alpha \in S.\, \exists v.\, \varphi^L = \alpha^L \wedge \varphi^P = \alpha^P[x \mapsto v]\})]\ x := \mathit{nonDet}()\ [P]$$

$$\textsc{Assume}\ \vdash [\lambda S.\, P(\{\varphi \mid \varphi \in S \wedge b(\varphi^P)\})]\ \textbf{assume}\ b\ [P]$$

$$\textsc{Assign}\ \vdash [\lambda S.\, P(\{\varphi \mid \exists \alpha \in S.\, \varphi^L = \alpha^L \wedge \varphi^P = \alpha^P[x \mapsto e(\varphi^P)]\})]\ x := e\ [P] \qquad \textsc{Skip}\ \vdash [P]\ \textbf{skip}\ [P]$$

Figure 5.2.: Core rules of Hyper Hoare Logic. The meaning of the operators $\otimes$ and $\otimes_{n\in\mathbb{N}}$ are defined in Definition 5.4.1 and Definition 5.4.2, respectively.

will derive in Section 5.5 syntactic versions of these rules, which are less expressive, but easier to apply.

The rule Choice (for non-deterministic choice) is more involved. Most standard Hoare logics use the same assertion Q as postcondition of all three triples. However, such a rule would not be sound in Hyper Hoare Logic. Consider for instance an application of this hypothetical Choice rule where both P and Q are defined as $\lambda S.\, |S| = 1$, expressing that there is a single pre- and post-state. If commands C_1 and C_2 are deterministic, the antecedents of the rule can be proved because a single pre-state leads to a single post-state. However, the non-deterministic choice will in general produce *two* post-states, such that the postcondition is violated.

To account for the effects of non-determinism on the sets of states described by hyper-assertions, we obtain the postcondition of the non-deterministic choice by combining the postconditions of its branches. As shown by Lemma 5.3.1 (point 6), executing the non-deterministic choice $C_1 + C_2$ in the set of states S amounts to executing C_1 in S and C_2 in S, and taking the union of the two resulting sets of states. Thus, if $Q_1(\mathit{sem}(C_1, S))$ and $Q_2(\mathit{sem}(C_2, S))$ hold then the postcondition of $C_1 + C_2$ must characterize the union $\mathit{sem}(C_1, S) \cup \mathit{sem}(C_2, S)$ The postcondition of the rule Choice, $Q_1 \otimes Q_2$, achieves that:

Definition 5.4.1 Union of hyper-assertions.
A set S satisfies $Q_1 \otimes Q_2$ iff it can be split into two (potentially overlapping) sets S_1 and S_2 (the sets of post-states of the branches), such that S_1 satisfies Q_1 and S_2 satisfies Q_2:

$$(Q_1 \otimes Q_2)(S) \triangleq (\exists S_1, S_2.\, S = S_1 \cup S_2 \wedge Q_1(S_1) \wedge Q_2(S_2))$$

The rule Iter for non-deterministic iterations generalizes our treatment of non-deterministic choice. It employs an indexed loop invariant I, which maps a natural number n to a hyper-assertion I_n. I_n characterizes the set of states reached after executing n times the command C in a set of initial states that satisfies I_0. Analogously to the rule Choice, the indexed

invariant avoids using the same hyper-assertion for all non-deterministic choices. The precondition of the rule's conclusion and its premise allow us to prove (inductively) that the triple $[I_0]\ C^n\ [I_n]$ holds for all n. I_n thus characterizes the set of reachable states after exactly n iterations of the loop. Since our loop is non-deterministic (*i.e.*, has no loop condition), the set of reachable states after the loop is the union of the sets of reachable states after each iteration. The postcondition of the conclusion captures this intuition, by using a version of the $\otimes$ operator generalized to an indexed family of hyper-assertions:

Definition 5.4.2 Indexed union of hyper-assertions.
A set S satisfies $\bigotimes_{n\in\mathbb{N}} I_n$ iff it can be split into $\bigcup_i f(i) = f(0)\cup\ldots\cup f(i)\cup\ldots$, where $f(i)$ (the set of reachable states after exactly i iterations) satisfies I_i (for each $i \in \mathbb{N}$):

$$(\textstyle\bigotimes_{n\in\mathbb{N}} I_n)(S) \triangleq \big(\exists f.\, (S = \bigcup_{n\in\mathbb{N}} f(n)) \wedge (\forall n \in \mathbb{N}.\, I_n(f(n)))\big)$$

Rules for terminating hyper-triples

While the rules presented in Figure 5.2 are designed for *partially correct* hyper-triples (as defined in Definition 5.3.5), they are all sound for *terminating hyper-triples* (as presented at the end of Section 5.3.2), except for the rule Assume, as **assume** b does not reach a final state when b does not hold. Formally, we have proven that, if the program C does not contain any **assume** statement, then the two types of hyper-triples are equivalent, *i.e.*, $\vdash [P]\ C\ [Q] \iff \vdash_{\Downarrow} [P]\ C\ [Q]$.

However, both if-statements and while loops contain **assume** statements, as

$$\textbf{if}\ (b)\ \{C_1\}\ \textbf{else}\ \{C_2\} \triangleq (\textbf{assume}\ b;\ C_1) + (\textbf{assume}\ \neg b;\ C_2)$$
$$\textbf{while}\ (b)\ \{C\} \triangleq (\textbf{assume}\ b;\ C)^*;\ \textbf{assume}\ \neg b$$

This is not a problem for establishing hyper-triples for if-statements, as the condition b is either true or false in any given state. Formally, we have proven the following rule sound, where the termination of each branch ensures the termination of the composed if-statement:

IfTot
$$\frac{\vdash [P]\ \textbf{assume}\ b\ [P_\top] \qquad \vdash [P]\ \textbf{assume}\ \neg b\ [P_\bot] \quad \vdash_{\Downarrow} [P_\top]\ C_1\ [Q_1] \quad \vdash_{\Downarrow} [P_\bot]\ C_2\ [Q_2]}{\vdash_{\Downarrow} [P]\ \textbf{if}\ (b)\ \{C_1\}\ \textbf{else}\ \{C_2\}\ [Q1 \otimes Q_2]}$$

Crucially, we decompose the statement **assume** b; C_1 into two parts: **assume** b, which might not terminate, and the command C_1, which is required to terminate (and similarly for **assume** $\neg b$; C_2).

For while loops, by combining the rules Iter, Assume, and Seq, we obtain the following rule for *partially correct* hyper-triples:

WhileDesugared
$$\frac{\vdash [I_n]\ \textbf{assume}\ b;\ C\ [I_{n+1}] \qquad \vdash [\textstyle\bigotimes_{n\in\mathbb{N}} I_n]\ \textbf{assume}\ \neg b\ [Q]}{\vdash [I_0]\ \textbf{while}\ (b)\ \{C\}\ [Q]}$$

The corresponding rule for *terminating hyper-triples* additionally requires proving that a loop variant strictly decreases after every iteration, as follows:

$$\textsc{WhileDesugaredTot}\quad \frac{\vdash [I_n]\ \textbf{assume}\ b\ [R_n] \qquad \vdash_{\Downarrow} [R_n \wedge \Box(e = t^L)]\ C\ [P_{n+1} \wedge \Box(e \prec t^L)] \qquad \vdash [\bigotimes_{n\in\mathbb{N}} I_n]\ \textbf{assume}\ \neg b\ [Q] \qquad t^L \notin fv(I_n) \cup fv(R_n) \qquad \prec \text{ well-founded}}{\vdash_{\Downarrow} [P_0]\ \textbf{while}\ (b)\ \{C\}\ [Q]}$$

In this rule, we use a fresh logical variable t^L to first record (in the precondition) the initial value of the ranking function e, which we then use (in the postcondition) to check that the ranking function e has indeed strictly decreased. Moreover, as in the rule IfTot, we split the statement **assume** b; C into **assume** b, which might not terminate, and the command C, which is required to terminate.

5.4.2. Soundness and Completeness

We have proved in Isabelle/HOL that Hyper Hoare Logic is sound and complete. That is, every hyper-triple that can be derived from the core rules is valid, and vice versa. Note that Figure 5.2 contains only the *core rules* of Hyper Hoare Logic. These are sufficient to prove completeness; all rules presented later in this chapter are only useful to make proofs more succinct and natural.

Theorem 5.4.1 Soundness of the core rules.
Hyper Hoare Logic is sound:

$$\textit{If} \vdash [P]\ C\ [Q]\ \textit{then} \models [P]\ C\ [Q].$$

Proof. We prove $\forall P, Q. \vdash [P]\ C\ [Q] \Rightarrow \models [P]\ C\ [Q]$ by straightforward structural induction on C. The cases for **skip**, C_1; C_2, $C_1 + C_2$, and C^*, directly follow from Lemma 5.3.1. □

Theorem 5.4.2 Completeness of the core rules.
Hyper Hoare Logic is complete:

$$\textit{If} \models [P]\ C\ [Q]\ \textit{then} \vdash [P]\ C\ [Q].$$

Proof. We prove $H(C) \triangleq (\forall P, Q. \models [P]\ C\ [Q] \Rightarrow \vdash [P]\ C\ [Q])$ by structural induction over C. We show the case for $C \triangleq C_1 + C_2$; the proof for the non-deterministic iteration is analogous, and the other cases are standard or straightforward.

We assume $H(C_1)$ and $H(C_2)$, and want to prove $H(C)$ where $C \triangleq C_1 + C_2$. As we illustrate after this proof sketch, we need to consider each possible value V of the set of extended states S separately. For an arbitrary value V, we define $P_V \triangleq (\lambda S.\, P(S) \wedge S = V)$, $R^1_V \triangleq (\lambda S.\, S = sem(C_1, V) \wedge P(V))$, and $R^2_V \triangleq (\lambda S.\, S = sem(C_2, V))$. We get $\vdash [P_V]\ C_1\ [R^1_V]$ from $H(C_1)$ and

$\vdash [P_V]\ C_2\ [R^2_V]$ from $H(C_2)$. We then construct the following derivation, using the core rules:

$$\dfrac{\dfrac{\dfrac{\forall V.\vdash [P_V]\ C_1\ [R^1_V] \qquad \forall V.\vdash [P_V]\ C_2\ [R^2_V]}{\forall V.\vdash [P_V]\ C_1 + C_2\ [R^1_V \otimes R^2_V]}\ \textsc{Choice}}{\vdash [\exists V.\, P_V]\ C\ [\exists V.\, R^1_V \otimes R^2_V]}\ \textsc{Exist} \qquad \dots}{\vdash [P]\ C_1 + C_2\ [Q]}\ \textsc{Cons}$$

To construct this derivation, we first apply the rule Choice, which gives us $\vdash [P_V]\ C\ [R^1_V \otimes R^2_V]$. Since we prove this triple for an arbitrary value V (that is, for all V), we then apply the rule Exist, to obtain $\vdash [\exists V.\, P_V]\ C\ [\exists V.\, R^1_V \otimes R^2_V]$. P clearly entails $\exists V.\, P_V$, and the postcondition $\exists V.\, R^1_V \otimes R^2_V$ entails $\lambda S.\, \exists V.\, P(V) \land S = sem(C_1, V) \cup sem(C_2, V)$, which precisely describes the sets of states $sem(C_1 + C_2, V)$ (Lemma 5.3.1(6)) where V satisfies P, and thus entails Q. By rule Cons, we get $\vdash [P]\ C\ [Q]$, which concludes the case. □

Note that our completeness theorem is not concerned with the expressivity of the assertion language because we use *semantic* hyper-assertions (*i.e.*, functions, see Definition 5.3.3). Similarly, by using semantic entailments in the rule Cons, we decouple the completeness of Hyper Hoare Logic from the completeness of the logic used to derive entailments.

Interestingly, the logic would *not* be complete without the core rule Exist, as we illustrate with the following simple example:

Example 5.4.1 Incompleteness of Hyper Hoare Logic without the rule Exist**.**
Let φ_v be the state that maps x to v and all other variables to 0. Let $P_v \triangleq (\lambda S.\, S = \{\varphi_v\})$. Clearly, the hyper-triples $[P_0]$ **skip** $[P_0]$, $[P_2]$ **skip** $[P_2]$, $[P_0]\ x := x + 1\ [P_1]$, and $[P_2]\ x := x + 1\ [P_3]$ are all valid. We would like to prove the hyper-triple $[P_0 \lor P_2]$ **skip** $+$ $(x := x + 1)$ $[\lambda S.\, S = \{\varphi_0, \varphi_1\} \lor S = \{\varphi_2, \varphi_3\}]$. That is, either P_0 holds before, and then we have $S = \{\varphi_0, \varphi_1\}$ afterwards, or P_2 holds before, and then we have $S = \{\varphi_2, \varphi_3\}$ afterwards. However, using the rule Choice only, the most precise triple we can prove is

$$\dfrac{\vdash [P_0 \lor P_2]\ \textbf{skip}\ [P_0 \lor P_2] \qquad \vdash [P_0 \lor P_2]\ x := x + 1\ [P_1 \lor P_3]}{\vdash [P_0 \lor P_2]\ \textbf{skip} + (x := x + 1)\ [(P_0 \lor P_2) \otimes (P_1 \lor P_3)]}\ \textsc{Choice}$$

The postcondition $(P_0 \lor P_2) \otimes (P_1 \lor P_3)$ is equivalent to

$$(P_0 \otimes P_1) \lor (P_0 \otimes P_3) \lor (P_2 \otimes P_1) \lor (P_2 \otimes P_3)$$

i.e.,

$$\lambda S.\, S = \{\varphi_0, \varphi_1\} \lor S = \{\varphi_0, \varphi_3\} \lor S = \{\varphi_2, \varphi_1\} \lor S = \{\varphi_2, \varphi_3\}$$

We thus have two spurious disjuncts, $P_0 \otimes P_3$ (*i.e.*, $S = \{\varphi_0, \varphi_3\}$) and $P_2 \otimes P_1$ (*i.e.*, $S = \{\varphi_2, \varphi_1\}$).

This example shows that the rule Choice on its own is not precise enough for the logic to be complete; we need at least a *disjunction* rule to distinguish

the two cases P_0 and P_2. In general, however, there might be an infinite number of cases to consider, which is why we need the rule Exist. The premise of this rule allows us to *fix* a set of states S that satisfies some precondition P and to prove the most precise postcondition for the precondition $\lambda S'.\ S = S'$; combining these precise postconditions with an existential quantifier in the conclusion of the rule allows us to obtain the most precise postcondition for the precondition P.

For our example, we can use the rule Exist with a Boolean b that records whether P_0 or P_2 is satisfied initially, as follows:

$$\dfrac{\dfrac{\vdash [(b \Rightarrow P_0) \wedge (\neg b \Rightarrow P_2)]\ \textbf{skip}\ [(b \Rightarrow P_0) \wedge (\neg b \Rightarrow P_2)] \quad \vdash [(b \Rightarrow P_0) \wedge (\neg b \Rightarrow P_2)]\ x := x+1\ [(b \Rightarrow P_1) \wedge (\neg b \Rightarrow P_3)]}{\vdash [(b \Rightarrow P_0) \wedge (\neg b \Rightarrow P_2)]\ \textbf{skip} + (x := x+1)\ [((b \Rightarrow P_0) \wedge (\neg b \Rightarrow P_2)) \otimes ((b \Rightarrow P_1) \wedge (\neg b \Rightarrow P_3))]}\ \textsc{Choice}}{\vdash \underbrace{[\exists b.\ (b \Rightarrow P_0) \wedge (\neg b \Rightarrow P_2)]}_{= P_0 \vee P_2}\ \textbf{skip} + (x := x+1)\ \underbrace{[\exists b.\ ((b \Rightarrow P_0) \wedge (\neg b \Rightarrow P_2)) \otimes ((b \Rightarrow P_1) \wedge (\neg b \Rightarrow P_3))]}_{= (P_0 \otimes P_2) \vee (P_1 \otimes P_3)}}\ \textsc{Exist}$$

5.5. Syntactic Rules

The core rules presented in Section 5.4 are optimized for expressiveness: They are sufficient to prove *any* valid hyper-triple (Theorem 5.4.2), but not necessarily in the simplest way. In particular, the rules for atomic statements Assume, Assign, and Havoc require a set comprehension in the precondition, which is necessary when dealing with arbitrary semantic hyper-assertions. However, by imposing syntactic restrictions on hyper-assertions, we can derive simpler rules, as we show in this section. In Section 5.5.1, we define a syntax for hyper-assertions, in which the set of states occurs only as range of universal and existential quantifiers. As we have seen in Section 5.2 and formally show in Appendix A.3, this syntax is sufficient to capture many useful hyperproperties. Moreover, it allows us to derive simple rules for assignments (Section 5.5.2) and assume statements (Section 5.5.3). All rules presented in this section have been proven sound in Isabelle/HOL.

5.5.1. Syntactic Hyper-Assertions

We define a restricted class of *syntactic hyper-assertions*, which can interact with the set of states only through universal and existential quantification over its states:

Definition 5.5.1 Syntactic hyper-expressions and hyper-assertions.
Hyper-expressions *e are defined by the following syntax, where φ ranges over states, x over (program or logical) variables, y over quantified variables, c over literals, $\oplus$ over binary operators (such as $+, -, *$ for integers, $++$ for lists, etc.), and f denotes functions from values to values (such as len for lists):*

$$e ::= c \mid y \mid \varphi^P(x) \mid \varphi^L(x) \mid e \oplus e \mid f(e)$$

Syntactic hyper-assertions *A are defined by the following syntax, where e ranges over hyper-expressions, b over boolean literals, and $\geq$ over binary*

AssignS
$\vdash [\mathcal{A}^e_x\,[P]]\ x := e\ [P]$

HavocS
$\vdash [\mathcal{H}_x\,[P]]\ x := \mathit{nonDet}()\ [P]$

AssumeS
$\vdash [\Pi_b\,[P]]\ \textbf{assume}\ b\ [P]$

Figure 5.3.: Selected syntactic rules of Hyper Hoare Logic. The syntactic transformations $\mathcal{A}^e_x\,[A]$, $\mathcal{H}_x\,[A]$, and $\Pi_b\,[A]$, are defined in Definition 5.5.3, Definition 5.5.4, and Definition 5.5.5, respectively.

operators (such as $=, \neq, <, >, \leq, \geq, \ldots$):

$$A ::= b \mid e \geq e \mid A \vee A \mid A \wedge A \mid \forall y.\, A \mid \exists y.\, A \mid \forall\langle\varphi\rangle.\, A \mid \exists\langle\varphi\rangle.\, A$$

Note that *hyper-expressions* are different from *program* expressions, since the latter can only refer to program variables of a *single* implicit state (*e.g.*, $x = y + z$), while the former can explicitly refer to different states (*e.g.*, $\varphi(x) = \varphi'(x)$). Negation $\neg A$ is defined recursively in the standard way. We also define $(A \Rightarrow B) \triangleq (\neg A \vee B)$, $\mathit{emp} \triangleq (\forall\langle\varphi\rangle.\, \bot)$, and $\Box p \triangleq (\forall\langle\varphi\rangle.\, p(\varphi))$, where p is a *state*[5] expression.

5: *State* expressions refer to a single (implicit) state. In contrast to program expressions, they may additionally refer to logical variables and use quantifiers over values.

The semantics of syntactic hyper-expressions and hyper-assertions is defined as follows:

Definition 5.5.2 Evaluation of syntactic hyper-expressions and satisfiability of hyper-assertions**.**
Let Σ a mapping from state variables (such as φ and φ') to states, and Δ a mapping from standard variables (such as x) to values.[6] *The evaluation of hyper-expressions is defined as follows:*

$$\begin{aligned}
[\![c]\!]^{\Sigma}_{\Delta} &\triangleq c\\
[\![y]\!]^{\Sigma}_{\Delta} &\triangleq \Delta(y)\\
[\![\varphi^P(x)]\!]^{\Sigma}_{\Delta} &\triangleq (\Sigma(\varphi))^P(x)\\
[\![\varphi^L(x)]\!]^{\Sigma}_{\Delta} &\triangleq (\Sigma(\varphi))^L(x)\\
[\![e_1 \oplus e_2]\!]^{\Sigma}_{\Delta} &\triangleq [\![e_1]\!]^{\Sigma}_{\Delta} \oplus [\![e_2]\!]^{\Sigma}_{\Delta}\\
[\![f(e)]\!]^{\Sigma}_{\Delta} &\triangleq f([\![e]\!]^{\Sigma}_{\Delta})
\end{aligned}$$

Let S be a set of states. The satisfiability of hyper-assertions is defined as follows:

$$\begin{aligned}
S, \Sigma, \Delta \models b &\triangleq b\\
S, \Sigma, \Delta \models e_1 \geq e_2 &\triangleq \left([\![e_1]\!]^{\Sigma}_{\Delta} \geq [\![e_2]\!]^{\Sigma}_{\Delta}\right)\\
S, \Sigma, \Delta \models A \wedge B &\triangleq (S, \Sigma, \Delta \models A \wedge S, \Sigma, \Delta \models B)\\
S, \Sigma, \Delta \models A \vee B &\triangleq (S, \Sigma, \Delta \models A \vee S, \Sigma, \Delta \models B)\\
S, \Sigma, \Delta \models \forall x.\, A &\triangleq (\forall v.\, S, \Sigma, \Delta[x \mapsto v] \models A)\\
S, \Sigma, \Delta \models \exists x.\, A &\triangleq (\exists v.\, S, \Sigma, \Delta[x \mapsto v] \models A)\\
S, \Sigma, \Delta \models \forall\langle\varphi\rangle.\, A &\triangleq \left(\forall \alpha \in S.\, S, \Sigma[\varphi \mapsto \alpha], \Delta \models A\right)\\
S, \Sigma, \Delta \models \exists\langle\varphi\rangle.\, A &\triangleq \left(\exists \alpha \in S.\, S, \Sigma[\varphi \mapsto \alpha], \Delta \models A\right)
\end{aligned}$$

When interpreting hyper-assertions in hyper-triples, we usually start with Δ and Σ being the empty mappings.[7]

6: In our Isabelle formalization, these mappings are actually lists, since we use De Bruijn indices [215].

7: An exception is when there is an explicit quantifier around the triple, such as in the premises for the rule *While*-$\exists$ from Figure 5.5.

5.5.2. Syntactic Rules for Deterministic and Non-Deterministic Assignments

In classical Hoare logic, we obtain the precondition of the rule for the assignment $x := e$ by substituting x by e in the postcondition. The Hyper Hoare Logic syntactic rule for assignments AssignS (Figure 5.3) generalizes this idea by repeatedly applying this substitution for *every quantified state*. This syntactic transformation, written $\mathcal{A}^e_x[\_]$ is defined below. As an example, for the assignment $x := y + z$ and postcondition $\exists\langle\varphi\rangle.\, \forall\langle\varphi'\rangle.\, \varphi(x) \leq \varphi'(x)$, we obtain the precondition

$$\begin{aligned}&\mathcal{A}^{y+z}_x\left[\exists\langle\varphi\rangle.\, \forall\langle\varphi'\rangle.\, \varphi(x) \leq \varphi'(x)\right]\\ =\ &\left(\exists\langle\varphi\rangle.\, \forall\langle\varphi'\rangle.\, \varphi(y) + \varphi(z) \leq \varphi'(y) + \varphi'(z)\right)\end{aligned}$$

Definition 5.5.3 Syntactic transformation for deterministic assignments.
$\mathcal{A}^e_x[A]$ yields the hyper-assertion A, where $\varphi(x)$ is syntactically substituted by $e(\varphi)$ for all (existentially or universally) quantified states φ:

$$\begin{aligned}
\mathcal{A}^e_x[b] &\triangleq b\\
\mathcal{A}^e_x[e_1 \geq e_2] &\triangleq e_1 \geq e_2\\
\mathcal{A}^e_x[A \wedge B] &\triangleq \mathcal{A}^e_x[A] \wedge \mathcal{A}^e_x[B]\\
\mathcal{A}^e_x[A \vee B] &\triangleq \mathcal{A}^e_x[A] \vee \mathcal{A}^e_x[B]\\
\mathcal{A}^e_x[\forall x.\, A] &\triangleq \forall x.\, \mathcal{A}^e_x[A]\\
\mathcal{A}^e_x[\exists x.\, A] &\triangleq \exists x.\, \mathcal{A}^e_x[A]\\
\mathcal{A}^e_x\left[\forall\langle\varphi\rangle.\, A\right] &\triangleq \left(\forall\langle\varphi\rangle.\, \mathcal{A}^e_x\left[A[e(\varphi)/\varphi(x)]\right]\right)\\
\mathcal{A}^e_x\left[\exists\langle\varphi\rangle.\, A\right] &\triangleq \left(\exists\langle\varphi\rangle.\, \mathcal{A}^e_x\left[A[e(\varphi)/\varphi(x)]\right]\right)
\end{aligned}$$

where $A[y/x]$ refers to the standard syntactic substitution of x by y.

Similarly, our syntactic rule for non-deterministic assignments HavocS substitutes every occurrence of $\varphi(x)$, for every quantified state φ, by a fresh quantified variable v. This variable is universally quantified for universally-quantified states, capturing the intuition that we must consider all possible assigned values. In contrast, v is existentially quantified for existentially-quantified states, because it is sufficient to find one suitable behavior of the non-deterministic assignment. As an example, for the non-deterministic assignment $x := \mathit{nonDet}()$ and the aforementioned postcondition, we obtain the precondition

$$\mathcal{H}_x\left[\exists\langle\varphi\rangle.\, \forall\langle\varphi'\rangle.\, \varphi(x) \leq \varphi'(x)\right] = \left(\exists\langle\varphi\rangle.\, \exists v.\, \forall\langle\varphi'\rangle.\, \forall v'.\, v \leq v'\right)$$

Definition 5.5.4 Syntactic transformation for non-deterministic assignments.
$\mathcal{H}_x[A]$ yields the hyper-assertion A where $\varphi(x)$ is syntactically substituted by a fresh quantified variable v, universally (resp. existentially) quantified for

universally (resp. existentially) quantified states:

$$
\begin{aligned}
\mathcal{H}_x\,[b] &\triangleq b \\
\mathcal{H}_x\,[e_1 \geq e_2] &\triangleq e_1 \geq e_2 \\
\mathcal{H}_x\,[A \wedge B] &\triangleq \mathcal{H}_x\,[A] \wedge \mathcal{H}_x\,[B] \\
\mathcal{H}_x\,[A \vee B] &\triangleq \mathcal{H}_x\,[A] \vee \mathcal{H}_x\,[B] \\
\mathcal{H}_x\,[\forall x.\, A] &\triangleq \forall x.\, \mathcal{H}_x\,[A] \\
\mathcal{H}_x\,[\exists x.\, A] &\triangleq \exists x.\, \mathcal{H}_x\,[A] \\
\mathcal{H}_x\,\big[\forall \langle \varphi \rangle.\, A\big] &\triangleq \big(\forall \langle \varphi \rangle.\, \forall v.\, \mathcal{H}_x\,\big[A[v/\varphi(x)]\big]\big) \\
\mathcal{H}_x\,\big[\exists \langle \varphi \rangle.\, A\big] &\triangleq \big(\exists \langle \varphi \rangle.\, \exists v.\, \mathcal{H}_x\,\big[A[v/\varphi(x)]\big]\big)
\end{aligned}
$$

5.5.3. Syntactic Rules for Assume Statements

Intuitively, **assume** b provides additional information when proving properties *for all* states, but imposes an additional requirement when proving *the existence* of a state. This intuition is captured by the rule AssumeS shown in Figure 5.3. The syntactic transformation Π_b adds the state expression b as an assumption for universally-quantified states, and as a proof obligation for existentially-quantified states. As an example, for the statement **assume** $x \geq 0$ and the postcondition $\forall \langle \varphi \rangle.\, \exists \langle \varphi' \rangle.\, \varphi(x) \leq \varphi'(x)$, we obtain the precondition

$$
\begin{aligned}
&\Pi_{x \geq 0}\,\big[\forall \langle \varphi \rangle.\, \exists \langle \varphi' \rangle.\, \varphi(x) \leq \varphi'(x)\big] \\
&= \big(\forall \langle \varphi \rangle.\, \varphi(x) \geq 0 \Rightarrow (\exists \langle \varphi' \rangle.\, \varphi'(x) \geq 0 \wedge \varphi(x) \leq \varphi'(x))\big)
\end{aligned}
$$

Definition 5.5.5 Syntactic transformation for assume statements.
$\Pi_b\,[A]$ yields the hyper-assertion A where b is syntactically added as an assumption for universally-quantified states, and as a proof obligation for existentially-quantified states:

$$
\begin{aligned}
\Pi_p\,[b] &\triangleq b \\
\Pi_p\,[e_1 \geq e_2] &\triangleq e_1 \geq e_2 \\
\Pi_p\,[A \wedge B] &\triangleq \Pi_p\,[A] \wedge \Pi_p\,[B] \\
\Pi_p\,[A \vee B] &\triangleq \Pi_p\,[A] \vee \Pi_p\,[B] \\
\Pi_p\,[\forall x.\, A] &\triangleq \forall x.\, \Pi_p\,[A] \\
\Pi_p\,[\exists x.\, A] &\triangleq \exists x.\, \Pi_p\,[A] \\
\Pi_p\,\big[\forall \langle \varphi \rangle.\, A\big] &\triangleq \forall \langle \varphi \rangle.\, p(\varphi) \Rightarrow \Pi_p\,[A] \\
\Pi_p\,\big[\exists \langle \varphi \rangle.\, A\big] &\triangleq \exists \langle \varphi \rangle.\, p(\varphi) \wedge \Pi_p\,[A]
\end{aligned}
$$

We now illustrate the use of our three syntactic rules on the following example.

Example 5.5.1 Using the syntactic rules to prove a violation of GNI.
Figure 5.4 shows a proof outline that the program

$$C_4 \triangleq (y := \mathit{nonDet}();\ \textbf{assume}\ y \leq 9;\ o := h + y)$$

$\{\exists\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \varphi_1(h) \neq \varphi_2(h)\}$

$\{\exists\langle\varphi_1\rangle.\ (\exists\langle\varphi_2\rangle.\ (\forall\langle\varphi\rangle.\ \forall v.\ v \leq 9 \Rightarrow (\varphi(h) = \varphi_1(h) \Rightarrow \varphi_2(h) + 9 > \varphi(h) + v)))\}$ (Cons)

$\{\exists\langle\varphi_1\rangle.\ \exists v_1.\ v_1 \leq 9 \wedge (\exists\langle\varphi_2\rangle.\ \exists v_2.\ v_2 \leq 9 \wedge (\forall\langle\varphi\rangle.\ \forall v.\ v \leq 9 \Rightarrow ((\varphi(h) \neq \varphi_1(h)) \vee (\varphi(h) + v \neq \varphi_2(h) + v_2))))\}$ (Cons)

$y \coloneqq \mathit{nonDet}();$

$\{\exists\langle\varphi_1\rangle.\ \varphi_1(y) \leq 9 \wedge (\exists\langle\varphi_2\rangle.\ \varphi_2(y) \leq 9 \wedge (\forall\langle\varphi\rangle.\ \varphi(y) \leq 9 \Rightarrow (\varphi(h) \neq \varphi_1(h) \vee \varphi(h) + \varphi(y) \neq \varphi_2(h) + \varphi_2(y))))\}$ (HavocS)

assume $y \leq 9$;

$\{\exists\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \forall\langle\varphi\rangle.\ \varphi(h) \neq \varphi_1(h) \vee \varphi(h) + \varphi(y) \neq \varphi_2(h) + \varphi_2(y)\}$ (AssumeS)

$o \coloneqq h + y$

$\{\exists\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \forall\langle\varphi\rangle.\ \varphi(h) \neq \varphi_1(h) \vee \varphi(o) \neq \varphi_2(o)\}$ (AssignS)

Figure 5.4.: Proof outline showing that the program *violates* generalized non-interference. The rules used at each step of the derivation are shown on the right (the use of rule Seq is implicit).

from Example 5.2.7 violates GNI. This program leaks information about the secret h through its public output o because the pad it uses (variable y) is upper bounded. From the output o, we can derive a lower bound for the secret value of h, namely $h \geq o - 9$.

To see why C_4 violates GNI, consider two executions with different secret values for h, and where the execution for the larger secret value sets y to exactly 9. This execution will produce a larger public output l (since the other execution adds at most 9 to its smaller secret). Hence, these executions can be *distinguished* by their public outputs.

Our proof outline in Figure 5.4 captures this intuitive reasoning in a natural way. We start with the postcondition that corresponds to the negation of GNI, and work our way backward, by successively applying our syntactic rules AssignS, AssumeS, and HavocS. We conclude using the rule Cons: Since the precondition implies the existence of two states with different values for h, we first instantiate φ_1 and φ_2 such that φ_1 and φ_2 are both members of the set of initial states, and $\varphi_2(h) > \varphi_1(h)$.[8] We then instantiate $v_2 = 9$, such that, for any $v \leq 9$, $\varphi_2(h) + v_2 > \varphi(h) + v$, which concludes the proof.

8: Note that the quantified states φ_1, φ_2 and φ from different hyper-assertions can be unrelated. That is, the witnesses for φ_1 and φ_2 in the first hyper-assertion $\exists\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \varphi_1(h) \neq \varphi_2(h)$ are not necessarily the same as the ones in the second hyper-assertion $\exists\langle\varphi_1\rangle.\ \exists\langle\varphi_2\rangle.\ \varphi_2(h) > \varphi_1(h)$, which is why the entailment holds.

5.6. Loop Rules

To reason about standard while loops, we can derive from the core rule Iter in Figure 5.2 the rule WhileDesugared, shown in Figure 5.5 (recall that **while** (b) $\{C\} \triangleq$ (**assume** b; $C)^*$; **assume** $\neg b$). While this derived rule is expressive, it has two main drawbacks for usability: (1) Because of the use of the infinitary $\bigotimes_{n\in\mathbb{N}}$, it requires non-trivial *semantic* reasoning (via the consequence rule), and (2) the invariant I_n relates only the executions that perform *at least* n iterations, but ignores executions that perform fewer.

Example 5.6.1 The rule WhileDesugared is limited.
To illustrate problem (2), imagine that we want to prove that the hyper-assertion $\mathit{low}(l) \triangleq (\forall\langle\varphi\rangle.\ \forall\langle\varphi'\rangle.\ \varphi(l) = \varphi'(l))$ holds after a while loop. A natural choice for our loop invariant I_n would be $I_n \triangleq \mathit{low}(l)$ (independent of n). However, this invariant does *not* entail our desired postcondition $\mathit{low}(l)$. Indeed, $\bigotimes_{n\in\mathbb{N}} \mathit{low}(l)$ holds for a set of states iff

$$\textsc{WhileDesugared}\quad \frac{\vdash [I_n]\ \textbf{assume}\ b; C\ [I_{n+1}] \qquad \vdash [\textstyle\bigotimes_{n\in\mathbb{N}} I_n]\ \textbf{assume}\ \neg b\ [Q]}{\vdash [I_0]\ \textbf{while}\ (b)\ \{C\}\ [Q]}$$

$$\textsc{WhileSync}\quad \frac{I \models \mathit{low}(b) \qquad \vdash [I \wedge \Box b]\ C\ [I]}{\vdash [I]\ \textbf{while}\ (b)\ \{C\}\ [(I \vee \mathit{emp}) \wedge \Box(\neg b)]} \qquad \textsc{IfSync}\quad \frac{P \models \mathit{low}(b) \qquad \vdash [P \wedge \Box b]\ C_1\ [Q] \qquad \vdash [P \wedge \Box(\neg b)]\ C_2\ [Q]}{\vdash [P]\ \textbf{if}\ (b)\ \{C_1\}\ \textbf{else}\ \{C_2\}\ [Q]}$$

$$\textsc{While-}\forall^*\exists^*\quad \frac{\vdash [I]\ \textbf{if}\ (b)\ \{C\}\ [I] \qquad \vdash [I]\ \textbf{assume}\ \neg b\ [Q] \qquad \text{no } \forall\langle\_\rangle \text{ after any } \exists \text{ in } Q}{\vdash [I]\ \textbf{while}\ (b)\ \{C\}\ [Q]}$$

$$\textsc{While-}\exists\quad \frac{\forall v.\ \vdash [\exists\langle\varphi\rangle.\ P_\varphi \wedge b(\varphi) \wedge v = e(\varphi)]\ \textbf{if}\ (b)\ \{C\}\ [\exists\langle\varphi\rangle.\ P_\varphi \wedge e(\varphi) \prec v] \qquad \forall\varphi.\ \vdash [P_\varphi]\ \textbf{while}\ (b)\ \{C\}\ [Q_\varphi] \qquad \prec \text{ wf}}{\vdash [\exists\langle\varphi\rangle.\ P_\varphi]\ \textbf{while}\ (b)\ \{C\}\ [\exists\langle\varphi\rangle.\ Q_\varphi]}$$

Figure 5.5.: Hyper Hoare Logic rules for while loops (and branching). Recall that $\mathit{low}(b) \triangleq (\forall\langle\varphi\rangle, \langle\varphi'\rangle.\ b(\varphi) = b(\varphi'))$ $\Box b \triangleq (\forall\langle\varphi\rangle.\ b(\varphi))$, and $\mathit{emp} \triangleq (\forall\langle\varphi\rangle.\ \bot)$. In the rule While-$\exists$, $\prec$ must be *well-founded* (wf).

it is a *union of* sets of states that all *individually* satisfy $\mathit{low}(l)$. This property holds trivially in our example (simply choose one set per possible value of l) and, in particular, does not express that the entire set of states after the loop satisfies $\mathit{low}(l)$. Note that this does not contradict completeness (Theorem 5.4.2), but simply means that a stronger invariant I_n is needed.

In this section, we thus present three more convenient loop rules, shown in Figure 5.5, which capture powerful reasoning principles, and overcome those limitations: The rule WhileSync (Section 5.6.1) is the easiest to use, and can be applied whenever all executions of the loop have the same control flow. Two additional rules for while loops can be applied whenever the control flow differs. The rule While-$\forall^*\exists^*$ (Section 5.6.2) supports $\forall^*\exists^*$-postconditions, while the rule While-$\exists$ (Section 5.6.3) handles postconditions with a top-level existential quantifier. In our experience, these loop rules cover all practical hyper-assertions that can be expressed in our syntax. We are not aware of any practical program hyperproperty that requires multiple quantifier alternations.

5.6.1. Synchronized Control Flow

Standard loop invariants are sound in relational logics if all executions exit the loop *simultaneously* [103, 216]. In our logic, this synchronized control flow can be enforced by requiring that the loop guard b has the same value in all states (1) before the loop and (2) after every loop iteration, as shown by the rule WhileSync in Figure 5.5. After the loop, we get to assume $(I \vee \mathit{emp}) \wedge \Box(\neg b)$. That is, the loop guard b is false in all executions, and the invariant I holds, or the set of states is empty. The emp disjunct corresponds to the case where the loop does not terminate (*i.e.*, *no* execution terminates). The rule WhileSync is suitable for invariants I of the form $\forall^+\exists^*$, as in this case $I \vee \mathit{emp}$ is equivalent to I. Going back to our motivating example (Example 5.6.1), the natural invariant $I \triangleq \mathit{low}(l)$ with the rule WhileSync is now sufficient for our example, since we get the postcondition $(\mathit{low}(l) \vee \mathit{emp}) \wedge \Box(\neg b)$, which implies our desired (universally-quantified) postcondition $\mathit{low}(l)$.

[103]: Benton (2004), *Simple Relational Correctness Proofs for Static Analyses and Program Transformations*
[216]: Terauchi et al. (2005), *Secure Information Flow as a Safety Problem*

Synchronized loop rule for terminating hyper-triples

The rule WhileSync in Figure 5.5 is unsuitable for invariants I of the form $\exists^+\forall^*$, as $I \vee \mathit{emp}$ does *not* imply I for such I. In this case, it is necessary to prove that *at least one* execution of the loop terminates.

Using terminating hyper-triples, we have proven sound the following rule, which overcomes the aforementioned limitation:

$$\frac{\begin{array}{c}\textsc{WhileSyncTot}\\ \vdash_{\Downarrow} [I \wedge \Box(b \wedge e = t^L)]\ C\ [I \wedge \mathit{low}(b) \wedge \Box(e \prec t^L)] \\ \prec \text{ well-founded} \qquad t^L \notin \mathit{fv}(I)\end{array}}{\vdash_{\Downarrow} [I \wedge \mathit{low}(b)]\ \textbf{while}\ (b)\ \{C\}\ [I \wedge \Box(\neg b)]}$$

Unlike the rule *WhileSync*, the rule *WhileSyncTot* does not have the emp disjunct in the postcondition of its conclusion anymore, and thus can be used to prove hyperproperties of the form $\exists^+\forall^*$. It achieves this by requiring that (1) the loop body C terminates (using the terminating hyper-triples defined at the end of Section 5.3.2), and (2) that the loop itself terminates, by requiring that a variant e decreases in all executions. The initial value of the variant e is stored in the logical variable t^L, such that it can be referred to in the postcondition.

We also provide a rule for if statements with synchronized control flow (rule IfSync in Figure 5.5), which can be applied when all executions take the same branch. This rule is simpler to apply than the core rule Choice, since it avoids the $\otimes$ operator, which usually requires semantic reasoning.

Example 5.6.2 Using the rule WhileSync to prove GNI.
The program in Figure 5.6 takes as input a list h of secret values (but whose length is public), computes its prefix sum $[h[0], h[0]+h[1], \ldots]$, and encrypts the result by performing a one-time pad on each element of this prefix sum, resulting in the output $[h[0] \oplus k_0, (h[0] + h[1]) \oplus k_1, \ldots]$. The keys $k_0, k_1, \ldots$ are chosen non-deterministically at each iteration, via the variable k.[9]

Our goal is to prove that the encrypted output l does not leak information about the secret elements of h, provided that the attacker does not have any information about the non-deterministically chosen keys. We achieve this by formally proving that this program satisfies GNI. Since the length of the list h is public, we start with the precondition $\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \mathit{len}(\varphi_1(h)) = \mathit{len}(\varphi_2(h))$. This implies that all our executions will perform the same number of loop iterations. Thus, we use the rule WhileSync, with the natural loop invariant $I \triangleq (\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \varphi_1(i) = \varphi_2(i) \wedge \mathit{len}(\varphi_1(h)) = \mathit{len}(\varphi_2(h)) \wedge (\exists\langle\varphi\rangle.\ \varphi(h) = \varphi_1(h) \wedge \varphi(l) = \varphi_2(l)))$. The last conjunct corresponds to the postcondition we want to prove, while the former entails $\mathit{low}(i < \mathit{len}(h))$, as required by the rule WhileSync.

The proof of the loop body starts at the end with the loop invariant I, and works backward, using the syntactic rules HavocS and AssignS. From $I \wedge \Box(i < \mathit{len}(h))$, we have to prove that there exists a value v

9: In practice, the keys used in this program should be stored somewhere, so that one is later able to decrypt the output.

$\{\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ len(\varphi_1(h)) = len(\varphi_2(h))\}$

$\{\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ 0 = 0 \land len(\varphi_1(h)) = len(\varphi_2(h)) \land (\exists\langle\varphi\rangle.\ \varphi(h) = \varphi_1(h) \land [] = [])\}$ (Cons)

$s := 0$

$l := []$

$i := 0$

$\{\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \varphi_1(i) = \varphi_2(i) \land len(\varphi_1(h)) = len(\varphi_2(h)) \land (\exists\langle\varphi\rangle.\ \varphi(h) = \varphi_1(h) \land \varphi(l) = \varphi_2(l))\}$ (AssignS)

while $(i < len(h))$ {

$\{(\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \varphi_1(i) = \varphi_2(i) \land len(\varphi_1(h)) = len(\varphi_2(h)) \land (\exists\langle\varphi\rangle.\ \varphi(h) = \varphi_1(h) \land \varphi(l) = \varphi_2(l))) \land \square(i < len(h))\}$

$\{\forall\langle\varphi_1\rangle.\ \forall v_1.\ \forall\langle\varphi_2\rangle.\ \forall v_2.\ \varphi_1(i) + 1 = \varphi_2(i) + 1 \land len(\varphi_1(h)) = len(\varphi_2(h)) \land$
$(\exists\langle\varphi\rangle.\ \exists v.\ \varphi(h) = \varphi_1(h) \land \varphi(l) \mathbin{+\!\!+} [(\varphi(s) + \varphi(h)[\varphi(i)]) \oplus v] = \varphi_2(l) \mathbin{+\!\!+} [(\varphi_2(s) + \varphi_2(h)[\varphi_2(i)]) \oplus v_2])\}$ (Cons)

$s := s + h[i];$

$k := nonDet();$

$l := l \mathbin{+\!\!+} [s \oplus k];$

$i := i + 1;$

$\{\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \varphi_1(i) = \varphi_2(i) \land len(\varphi_1(h)) = len(\varphi_2(h)) \land (\exists\langle\varphi\rangle.\ \varphi(h) = \varphi_1(h) \land \varphi(l) = \varphi_2(l))\}$ (HavocS, AssignS)

}

$\{((\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \varphi_1(i) = \varphi_2(i) \land len(\varphi_1(h)) = len(\varphi_2(h)) \land (\exists\langle\varphi\rangle.\ \varphi(h) = \varphi_1(h) \land \varphi(l) = \varphi_2(l))) \lor emp) \land \square(i \geq len(h))\}$ (WhileSync)

$\{\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \exists\langle\varphi\rangle.\ \varphi(h) = \varphi_1(h) \land \varphi(l) = \varphi_2(l)\}$ (Cons)

Figure 5.6.: A proof that the program in black satisfies generalized non-interference (where the elements of list h are secret, but its length is public), using the rule WhileSync. [] represents the empty list, $+\!\!+$ represents list concatenation, $h[i]$ represents the i-th element of list h, and $\oplus$ represents the XOR operator.

such that

$\varphi(l) \mathbin{+\!\!+} [(\varphi(s)+\varphi(h)[\varphi(i)])\oplus v] = \varphi_2(l) \mathbin{+\!\!+} [(\varphi_2(s)+\varphi_2(h)[\varphi_2(i)])\oplus v_2]$

Since $\varphi(l) = \varphi_2(l)$, this boils down to proving that

$$(\varphi(s) + \varphi(h)[\varphi(i)]) \oplus v = (\varphi_2(s) + \varphi_2(h)[\varphi_2(i)]) \oplus v_2$$

which we achieve by choosing

$$v \triangleq (\varphi_2(s) + \varphi_2(h)[\varphi_2(i)]) \oplus v_2 \oplus (\varphi(s) + \varphi(h)[\varphi(i)])$$

5.6.2. $\forall^*\exists^*$-Hyperproperties

Let us now turn to the more general case, where different executions might exit the loop at different iterations. As explained at the start of this section, the main usability issue of the rule WhileDesugared is the precondition $\bigotimes_{n\in\mathbb{N}} I_n$ in the second premise, which requires non-trivial semantic reasoning. The $\bigotimes_{n\in\mathbb{N}}$ operator is required, because I_n ignores executions that exited the loop earlier; it relates only the executions that have performed *at least* n iterations. In particular, it would be unsound to replace the precondition $\bigotimes_{n\in\mathbb{N}} I_n$ by $\exists n.\ I_n$.

The rule While-$\forall^*\exists^*$ in Figure 5.5 solves this problem for the general case of $\forall^*\exists^*$-postconditions. The key insight is to reason about the successive

```
a := 0;
b := 1;
i := 0;
while (i < n)
{
    tmp := b;
    b := a + b;
    a := tmp;
    i := i + 1
}
```

(a) The program C_{fib}, which computes the n-th Fibonacci number.

```
x := 0;
y := 0;
i := 0;
while (i < k)
{
    r := nonDet();
    assume r ≥ 2;
    t := x;
    x := 2 * x + r;
    y := y + t * r;
    i := i + 1
}
```

(b) A program with a final state with minimal values for x and y.

Figure 5.7.: Two simple programs with loops, illustrating the need for the rules While-$\forall^*\exists^*$ and While-$\exists$.

unrollings of the while loop: The rule requires to prove an invariant I for the conditional statement **if** (b) $\{C\}$, in contrast to **assume** b; C in the rule WhileDesugared. This allows the invariant I to refer to *all* executions, *i.e.*, executions that are still running the loop (which will execute C), and executions that have already exited the loop (which will not execute C).

Example 5.6.3 Using the rule While-$\forall^*\exists^*$ to prove monotonicity of Fibonacci**.**
The program C_{fib} in Figure 5.7a takes as input an integer $n \geq 0$ and computes the n-th Fibonacci number (in variable a). We want to prove that C_{fib} is monotonic, *i.e.*, that the n-th Fibonacci number is greater than or equal to the m-th Fibonacci number whenever $n \geq m$, without making explicit what C_{fib} computes. Formally, we want to prove the hyper-triple

$$\begin{array}{c}[\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \varphi_1(t){=}1 \land \varphi_2(t){=}2 \Rightarrow \varphi_1(n) \geq \varphi_2(n)] \\ C_{fib} \\ [\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \varphi_1(t){=}1 \land \varphi_2(t){=}2 \Rightarrow \varphi_1(a) \geq \varphi_2(a)]\end{array}$$

where t is a logical variable used to track the execution (as explained in Section 5.2.2).

Intuitively, this program is monotonic because both executions will perform at least $\varphi_2(n)$ iterations, during which they will have the same values for a and b. The first execution will then perform $\varphi_1(n) - \varphi_2(n)$ additional iterations, during which a and b will increase, thus resulting in larger values for a and b.

We cannot use the rule WhileSync to make this intuitive argument formal, since both executions might perform a different number of iterations. Moreover, we cannot express this intuitive argument with the rule WhileDesugared either, since the invariant I_k only relates executions that perform *at least* k *iterations*, as explained earlier: After the first $\varphi_2(n)$ iterations, the loop invariant I_k cannot refer to the values of a and b in

the second execution, since this execution has already exited the loop.

However, we can use the rule While-$\forall^*\exists^*$ to prove that C_{fib} is monotonic, with the intuitive loop invariant

$$I \triangleq \square(b \geq a \geq 0) \wedge (\forall \langle \varphi_1 \rangle, \langle \varphi_2 \rangle.\ \varphi_1(t)=1 \wedge \varphi_2(t)=2 \\ \Rightarrow (\varphi_1(n)-\varphi_1(i) \geq \varphi_2(n)-\varphi_2(i) \wedge \varphi_1(a) \geq \varphi_2(a) \wedge \varphi_1(b) \geq \varphi_2(b)))$$

The second part captures the relation between the two executions: a and b are larger in the first execution than in the second one, and the first execution does at least as many iterations as the second one. The first part $\square(b \geq a \geq 0)$ is needed to prove that the additional iterations lead to larger values for a and b. The proof of this example is in the appendix (Appendix A.4.1).

Restriction to $\forall^*\exists^*$-hyperproperties. The rule While-$\forall^*\exists^*$ is quite general and powerful, since it can be applied to prove any postcondition of the shape $\forall^*\exists^*$, which includes *all* safety hyperproperties, as well as some liveness hyperproperties such as GNI. However, it cannot be applied for postconditions with a top-level existential quantification over states or values,[10] because this would be unsound, as illustrated by the following example.

10: If a syntactic hyper-assertion Q satisfies this *syntactic* restriction, then it satisfies the following *semantic* property, which we use to prove the soundness of the rule While-$\forall^*\exists^*$: For any non-decreasing sequence $(S_n)_{n\in\mathbb{N}}$ of set of states, if $\forall n.\ S_n \models Q$, then $(\bigcup_n S_n) \models Q$. The proof of Theorem 6.4.1, which establishes the soundness of a rule similar to While-$\forall^*\exists^*$, provides more explanations.

Example 5.6.4 The rule While-$\forall^*\exists^*$ without the syntactic restriction would be unsound.
Consider the program $C \triangleq$ **while** $(x < n)\ \{x := x+1\}$, which results in $x = n$ after the loop terminates (if $n \geq 0$). By *incorrectly* using the rule While-$\forall^*\exists^*$ with the loop invariant $I \triangleq \exists N.\ \forall \langle \varphi \rangle.\ \varphi(x) \leq N$ (which *violates* the syntactic restriction), we can construct the following *invalid* hyper-triple:

$$\dfrac{\dfrac{\cdots}{[I]\ \textbf{if}\ (x<n)\ \{x := x+1\}\ [I]} \qquad \dfrac{\cdots}{[I]\ \textbf{assume}\ \neg b\ [I]}}{[I]\ \textbf{while}\ (x<n)\ \{x := x+1\}\ [I]}\ \text{While-}\forall^*\exists^*$$

This (invalid) triple expresses that if we have an upper bound for the value of x in *all states* before the loop, then we have an upper bound for the value of x in *all states* after the loop. To see why it is invalid, consider for example the following set of states S, which has an upper bound for x ($N = 0$), but no upper bound for n:

$$S \triangleq \{\varphi \mid \exists n \geq 0.\ \varphi(n) = N \wedge \varphi(x) = 0\}$$

While S clearly satisfies the precondition, it leads to the following set of states after the loop, which has no upper bound for x (as $x = n$), and thus violates the postcondition:

$$sem(C, S) \triangleq \{\varphi \mid \exists n \geq 0.\ \varphi(n) = N \wedge \varphi(x) = N\}$$

The key issue is that the witness N that we get after n unrollings is not necessarily the same as the witness we get for $n+1$ unrollings, and thus, nothing guarantees that there is a *global* witness that works after *any* number of loop unrollings.

Similarly, using this rule with an invariant with a top-level existential quantification over *states* would also be unsound. Indeed, a triple such as $\vdash [\exists\langle\varphi\rangle.\, \forall\langle\varphi'\rangle.\, I]$ **if** (b) $\{C\}$ $[\exists\langle\varphi\rangle.\, \forall\langle\varphi'\rangle.\, I]$ implies that, for any n, there exists a state φ such that I holds for all states φ' reached after *unrolling the loop n times*. As in Example 5.6.4, the state φ might not be a valid witness for states φ' reached after *more than n loop unrollings*, and therefore we might have different witnesses for φ for each value of n. We thus have no guarantee that there is a *global* witness that works for all states φ' after *any* number of loop unrollings. To handle such examples, we present a rule for $\exists^*\forall^*$-hyperproperties next.

5.6.3. $\exists^*\forall^*$-Hyperproperties

The rule While-$\forall^*\exists^*$ can be applied for any postcondition of the form $\forall^*\exists^*$, which includes all safety hyperproperties as well as some liveness hyperproperties such as GNI, but cannot be applied to prove postconditions with a top-level existential quantifier, such as postconditions of the shape $\exists^*\forall^*$ (*e.g.*, to prove the existence of minimal executions, or to prove that a $\forall^*\exists^*$-hyperproperty is violated). In this case, we can apply the rule While-$\exists$ in Figure 5.5. To the best of our knowledge, this is the first program logic rule that can deal with $\exists^*\forall^*$-hyperproperties for loops. This rule splits the reasoning into two parts: First, we prove that there is a *terminating* state φ such that the hyper-assertion P_φ holds after some number of loop unrollings. This is achieved via the first premise of the rule, which requires a well-founded relation $\prec$, and a variant $e(\varphi)$ that strictly decreases at each iteration, until $b(\varphi)$ becomes false and φ exits the loop.[11] In a second step, we fix the state φ (since it has exited the loop), which corresponds to our global witness, and prove $\vdash [P_\varphi]$ **while** (b) $\{C\}$ $[Q_\varphi]$ using any loop rule. For example, if P_φ has another top-level existential quantifier, we can apply the rule While-$\exists$ once more; if P_φ is a $\forall^*\exists^*$-hyper-assertion, we can apply the rule While-$\forall^*\exists^*$.

11: Interestingly, note that the existentially-quantified state φ in the postcondition of the first premise of the rule While-$\exists$ does *not* have to be from the same execution as the one in the precondition.

Example 5.6.5 Using the rule While-$\exists$ to prove the existence of minimal executions**.**
Consider proving that the program C_m in Figure 5.7b has a final state with a minimal value for x and y. Formally, we want to prove the triple

$$[\neg emp \wedge \square(k \geq 0)]\ C_m\ [\exists\langle\varphi\rangle.\, \forall\langle\alpha\rangle.\, \varphi(x) \leq \alpha(x) \wedge \varphi(y) \leq \alpha(y)]$$

Since the set of initial states is not empty and k is always non-negative, we know that there is an initial state with a minimal value for k. We prove that this state leads to a final state with minimal values for x and y, using the rule While-$\exists$. For the first premise, we choose the variant[12] $k - i$, and the invariant $P_\varphi \triangleq (\forall\langle\alpha\rangle.\, 0 \leq \varphi(x) \leq \alpha(x) \wedge 0 \leq \varphi(y) \leq \alpha(y) \wedge \varphi(k) \leq \alpha(k) \wedge \varphi(i) = \alpha(i))$, capturing both that φ has minimal values for x and y, but also that φ will be the first state to exit the loop. We prove that this is indeed an invariant for the loop, by choosing $r = 2$ for the non-deterministic assignment for φ. Finally, we prove the second premise with $Q_\varphi \triangleq (\forall\langle\alpha\rangle.\, 0 \leq \varphi(x) \leq \alpha(x) \wedge 0 \leq \varphi(y) \leq \alpha(y))$ and the rule While-$\forall^*\exists^*$. The proof of this example is in the appendix (Appendix A.4.2).

12: We interpret $\prec$ as $<$ between natural numbers, *i.e.*, $a \prec b$ iff $0 \leq a$ and $a < b$, which is well-founded.

5.7. Compositionality Rules

Since the core rules are complete (Theorem 5.4.2), they can be used to derive any valid hyper-triple, but they are limited in their *compositionality*, as shown by the following example.

Example 5.7.1 Composing generalized non-interference with non-interference.
Consider the sequential composition of a command C_1 that satisfies *generalized* non-interference (GNI) with a command C_2 that satisfies non-interference (NI). We would like to prove that $C_1; C_2$ satisfies GNI (the weaker property). As discussed in Section 5.2.3, a possible postcondition for C_1 is $GNI_l^h \triangleq (\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle. \exists\langle\varphi\rangle. \varphi_1^L(h) = \varphi^L(h) \land \varphi^P(l) = \varphi_2^P(l))$, while a possible precondition for C_2 is $low(l) \triangleq (\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle. \varphi_1(l) = \varphi_2(l))$. The corresponding hyper-triples for C_1 and C_2 cannot be composed using the core rules. In particular, rule Seq cannot be applied (even in combination with Cons), since the postcondition of C_1 does not imply the precondition of C_2.

Note that this observation does *not* contradict completeness: By Theorem 5.4.2, it is possible to prove *more precise* triples for C_1 and C_2, such that the postcondition of C_1 matches the precondition of C_2. However, to enable modular reasoning, our goal is to construct the proof by composing the given triples for the individual commands rather than deriving new ones.

In this section, we present *compositionality rules* for hyper-triples (Section 5.7.1). These rules are *admissible* in Hyper Hoare Logic, in the sense that they do not modify the set of valid hyper-triples that can be proved. Rather, these rules enable flexible compositions of hyper-triples (such as those discussed above). We illustrate these rules on two examples (Section 5.7.2): Composing minimality with monotonicity, and GNI with NI. All technical results presented in this section (soundness of the rules shown in Figure 5.8 and validity of the examples) have been formalized and proved in Isabelle/HOL.

5.7.1. Compositionality Rules

Figure 5.8 shows compositionality rules for Hyper Hoare Logic, which we discuss below.

Linking. To prove hyper-triples of the form $[\forall\langle\varphi_1\rangle. P_{\varphi_1}]\ C\ [\forall\langle\varphi_2\rangle. Q_{\varphi_2}]$, the rule Linking considers each pair of pre-state φ_1 and post-state φ_2 separately, and lets one assume that φ_2 can be reached by executing C in the state φ_1, and that logical variables do not change during this execution.

Conjunctions and disjunctions. Hyper Hoare Logic admits the usual rules for conjunction (And and Forall in Figure 5.8) and disjunction (Or in Figure 5.8 and the core rule Exist in Figure 5.2).

$$\textsc{Forall}\ \frac{\forall x.\ (\vdash [P_x]\ C\ [Q_x])}{\vdash [\forall x.\ P_x]\ C\ [\forall x.\ Q_x]} \qquad \textsc{Linking}\ \frac{\forall \varphi_1, \varphi_2.\ (\varphi_1^L = \varphi_2^L \land \vdash [\langle \varphi_1 \rangle]\ C\ [\langle \varphi_2 \rangle] \implies \vdash [P_{\varphi_1}]\ C\ [Q_{\varphi_2}])}{\vdash [\forall \langle \varphi \rangle.\ P_\varphi]\ C\ [\forall \langle \varphi \rangle.\ Q_\varphi]}$$

$$\textsc{IndexedUnion}\ \frac{\forall x.\ (\vdash [P_x]\ C\ [Q_x])}{\vdash [\bigotimes_{x \in X} P_x]\ C\ [\bigotimes_{x \in X} Q_x]} \qquad \textsc{Union}\ \frac{\vdash [P_1]\ C\ [Q_1] \qquad \vdash [P_2]\ C\ [Q_2]}{\vdash [P_1 \otimes P_2]\ C\ [Q_1 \otimes Q_2]} \qquad \textsc{BigUnion}\ \frac{\vdash [P]\ C\ [Q]}{\vdash [\bigotimes P]\ C\ [\bigotimes Q]}$$

$$\textsc{And}\ \frac{\vdash [P_1]\ C\ [Q_1] \qquad \vdash [P_2]\ C\ [Q_2]}{\vdash [P_1 \land P_2]\ C\ [Q_1 \land Q_2]} \qquad \textsc{Or}\ \frac{\vdash [P_1]\ C\ [Q_1] \qquad \vdash [P_2]\ C\ [Q_2]}{\vdash [P_1 \lor P_2]\ C\ [Q_1 \lor Q_2]} \qquad \textsc{AtMost}\ \frac{\vdash [P]\ C\ [Q]}{\vdash [\sqsubseteq P]\ C\ [\sqsubseteq Q]}$$

$$\textsc{AtLeast}\ \frac{\vdash [P]\ C\ [Q]}{\vdash [\sqsupseteq P]\ C\ [\sqsupseteq Q]} \qquad \textsc{LUpdateS}\ \frac{\vdash [P \land (\forall \langle \varphi \rangle.\ \varphi(t) = e(\varphi))]\ C\ [Q] \qquad t \notin fv(P) \cup fv(Q) \cup fv(e)}{\vdash [P]\ C\ [Q]}$$

$$\textsc{LUpdate}\ \frac{P \Rightarrow^V P' \qquad \vdash [P']\ C\ [Q] \qquad inv^V(Q)}{\vdash [P]\ C\ [Q]} \qquad \textsc{True}\ \vdash [P]\ C\ [\top] \qquad \textsc{False}\ \vdash [\bot]\ C\ [Q]$$

$$\textsc{FrameSafe}\ \frac{\vdash [P]\ C\ [Q] \qquad \text{no } \exists \langle \_ \rangle \text{ in } F \qquad wr(C) \cap fv(F) = \varnothing}{\vdash [P \land F]\ C\ [Q \land F]} \qquad \textsc{Specialize}\ \frac{\vdash [P]\ C\ [Q] \qquad wr(C) \cap fv(b) = \varnothing}{\vdash [\Pi_b\ [P]]\ C\ [\Pi_b\ [Q]]}$$

Figure 5.8.: Compositionality rules of Hyper Hoare Logic. All these rules have been proven sound in Isabelle/HOL. $wr(C)$ corresponds to the set of program variables that are potentially written by C (*i.e.*, that appear on the left-hand side of an assignment), while $fv(F)$ corresponds to the set of program variables that appear in look-up expressions for quantified states. For example, $fv(\forall \langle \varphi \rangle.\ \exists n.\ \varphi^P(x) = n^2) = \{x\}$. The rules FrameSafe and Specialize assume that F (in the former) and P and Q (in the latter) are *syntactic hyper-assertions*, *i.e.*, that they are written in the syntax presented in Section 5.5. In particular, the transformation $\Pi_b\ [P]$ is formally defined in Section 5.5. The operators $\bigotimes$, $\sqsubseteq$, and $\sqsupseteq$ are defined as follows: $\bigotimes P \triangleq (\lambda S.\ \exists F.\ (S = \bigcup_{S' \in F} S') \land (\forall S' \in F.\ P(S')))$, $\sqsubseteq P \triangleq (\lambda S.\ \exists S'.\ S \subseteq S' \land P(S'))$, and $\sqsupseteq P \triangleq (\lambda S.\ \exists S'.\ S' \subseteq S \Rightarrow P(S'))$.

Framing. Similarly to the frame rules in Hoare logic and separation logic [10], Hyper Hoare Logic admits rules that allow us to frame information about states that is not affected by the execution of C. The rule FrameSafe allows us to frame any *syntactic* hyper-assertion F if (1) it does not refer to variables that the program can modify, and (2) it does not existentially quantify over states.[13] While (1) is standard, (2) is specific to hyper-assertions: Framing the existence of a state (*e.g.*, with $F \triangleq (\exists \langle \varphi \rangle.\ \top)$) would be unsound if the execution of the program in the state φ does not terminate.

[10]: Reynolds (2002), *Separation Logic*

13: Semantically, condition (2) means that F is *downwards-closed*, *i.e.*, if it satisfied by a set S, then it must also be satisfied by all its subsets $S' \subseteq S$.

Framing for terminating hyper-triples

Restriction (2) of the rule FrameSafe can be lifted if we use *terminating hyper-triples* (as defined at the end of Section 5.3.2), as they guarantee that each initial state existentially quantified in the frame F has a corresponding final state. Formally, we have proven the following rule sound in Isabelle/HOL, which allows the syntactic hyper-assertion F to universally *and existentially* quantify over states:

$$\textsc{Frame}\ \frac{wr(C) \cap fv(F) = \varnothing}{\vdash_{\Downarrow} [P \land F]\ C\ [Q \land F]}$$

Crucially, the soundness of this rule relies on the fact that F is a *syntactic* hyper-assertion. Such a rule would be unsound for *semantic* hyper-assertions F. For example, using $F \triangleq (\lambda S.\, |S| = 1)$ (which does not mention any program variable) as the frame, we could prove the following *invalid* hyper-triple:

$$[\top \wedge F]\ x := \mathit{nonDet}()\ [\top \wedge F]$$

This triple is invalid, as the non-deterministic assignment results in more than one reachable state.

Decompositions. As explained at the beginning of this section, the two triples $[P]\ C_1\ [GNI_l^h]$ and $[\mathit{low}(l)]\ C_2\ [Q]$ cannot be composed because GNI_l^h does not entail $\mathit{low}(l)$ (not all states in the set S of final states of C_1 need to have the same value for l). However, we can prove GNI for the composed commands by decomposing S into subsets that all satisfy $\mathit{low}(l)$ and considering each subset separately. The rule BigUnion allows us to perform this decomposition (formally expressed with the hyper-assertion $\bigotimes \mathit{low}(l)$), use the specification of C_2 on each of these subsets (since they all satisfy the precondition of C_2), and eventually recompose the final set of states (again with the operator $\bigotimes$) to prove our desired postcondition. Hyper Hoare Logic also admits rules for binary unions (rule Union) and indexed unions (rule IndexedUnion).

Note that unions ($\otimes$ and $\bigotimes$) and disjunctions in hyper-assertions are very *different*: $(P \otimes Q)(S)$ expresses that the set S can be decomposed into two sets S_P (satisfying P) and S_Q (satisfying Q), while $(P \vee Q)(S)$ expresses that the entire set S satisfies P or Q. Similarly, intersections and conjunctions are very different: While Hyper Hoare Logic admits conjunction rules, rules based on intersections would be unsound, as shown by the following example:

Example 5.7.2 Unsoundness of a potential intersection rule.
Let $P_i \triangleq (\exists \langle \varphi \rangle.\, \varphi(x) = i) \wedge (\forall \langle \varphi \rangle.\, \varphi(x) = i)$.

Both triples $[P_1]\ x := 1\ [P_1]$ and $[P_2]\ x := 1\ [P_1]$ are valid. *If* we had an intersection rule, we could derive from these two triples the triple

$$\begin{gathered}[\lambda S.\, \exists S_1, S_2.\, S = S_1 \cap S_2 \wedge P_1(S_1) \wedge P_2(S_2)] \\ x := 1 \\ [\lambda S.\, \exists S_1, S_2.\, S = S_1 \cap S_2 \wedge P_1(S_1) \wedge P_1(S_2)]\end{gathered}$$

This triple is however *invalid*, as the precondition is satisfiable by the empty set, whereas the postcondition is not (as the postcondition is equivalent to P_1).

Specializing hyper-triples. By definition, a hyper-triple can only be applied to a set of states that satisfies its precondition, which can be restrictive. In cases where only a *subset* of the current set of states satisfies the precondition, one can obtain a *specialized* triple using the rule Specialize. This rule uses the syntactic transformation Π_b (Definition 5.5.5) to weaken both the precondition and the postcondition of the triple, which is sound as long as the validity of b is not influenced by executing C. Intuitively,

$\Pi_b\ [P]$ holds for a set S iff P holds for the subset of states from S that satisfy b.

Example 5.7.3 Specializing hyper-triples for proving monotonicity. From the triples $[\Box(x \geq 0)]\ C\ [\Box(y \geq 0)]$ and $[\Box(x \leq 0)]\ C\ [\Box(y \leq 0)]$, we can derive the following triple

$$[\Box(t{=}1 \Rightarrow x \geq 0) \wedge \Box(t{=}2 \Rightarrow x < 0)]$$
$$C$$
$$[\forall \langle \varphi_1 \rangle, \langle \varphi_2 \rangle.\ \varphi_1(t){=}1 \wedge \varphi_2(t){=}2 \Rightarrow \varphi_1(y) \geq \varphi_2(y)]$$

whose postcondition corresponds to $mono^t_y$ (Example 5.2.3), as follows:

$$\dfrac{\dfrac{[\Box(x \geq 0)]\ C\ [\Box(y \geq 0)]}{[\Box(t{=}1 \Rightarrow x \geq 0)]\ C\ [\Box(t{=}1 \Rightarrow y \geq 0)]}\ \textsc{Sp.} \quad \dfrac{[\Box(x \leq 0)]\ C\ [\Box(y \leq 0)]}{[\Box(t{=}2 \Rightarrow x \leq 0)]\ C\ [\Box(t{=}1 \Rightarrow y \leq 0)]}\ \textsc{Sp.}}{[\Box(t{=}1 \Rightarrow x \geq 0) \wedge \Box(t{=}2 \Rightarrow x \leq 0)]\ C\ [\Box(t{=}1 \Rightarrow y \geq 0) \wedge \Box(t{=}2 \Rightarrow y \leq 0)]}\ \textsc{And}$$
$$\dfrac{}{[\Box(t{=}1 \Rightarrow x \geq 0) \wedge \Box(t{=}2 \Rightarrow x < 0)]\ C\ [\forall \langle \varphi_1 \rangle, \langle \varphi_2 \rangle.\ \varphi_1(t){=}1 \wedge \varphi_2(t){=}2 \Rightarrow \varphi_1(y) \geq \varphi_2(y)]}\ \textsc{Cons}$$

Logical updates. Logical variables play an important role in the expressiveness of the logic: As we have informally shown in Section 5.2.2 and formally show in Appendix A.3, relational specifications are typically expressed in Hyper Hoare Logic by using logical variables to formally link the pre-state of an execution with the corresponding post-states. Since logical variables cannot be modified by the execution, these tags are preserved.

To apply this proof strategy with existing triples, it is often necessary to update logical variables to introduce such tags. The rule LUpdate allows us to update the logical variables in a set V, provided that (1) from every set of states S that satisfies P, we can obtain a new set of states S' that satisfies P', by only updating (for each state) the logical variables in V, (2) we can prove the triple with the updated set of initial states, and (3) the postcondition Q cannot distinguish between two sets of states that are equivalent up to logical variables in V. We formalize this intuition in the following:

Definition 5.7.1 Logical updates.
*Let V be a set of logical variable names. Two states φ_1 and φ_2 are **equal up to logical variables** V, written $\varphi_1 \overset{V}{=} \varphi_2$, iff*

$$\forall i.\ i \notin V \Rightarrow \varphi_1^L(i) = \varphi_2^L(i) \text{ and } \varphi_1^P = \varphi_2^P$$

*Two sets of states S_1 and S_2 are **equivalent up to logical variables** V, written $S_1 \overset{V}{=} S_2$, iff every state $\varphi_1 \in S_1$ has a corresponding state $\varphi_2 \in S_2$ with the same values for all variables except those in V, and vice-versa. Formally, $S_1 \overset{V}{=} S_2$ holds iff*

$$\forall \varphi_1 \in S_1.\ \exists \varphi_2 \in S_2.\ \varphi_1 \overset{V}{=} \varphi_2 \text{ and } \forall \varphi_2 \in S_2.\ \exists \varphi_1 \in S_1.\ \varphi_1 \overset{V}{=} \varphi_2$$

*A hyper-assertion P **entails** a hyper-assertion P' **modulo logical variables***

$$\dfrac{\dfrac{\vdash [\mathit{singleton}]\ C_2\ [\mathit{singleton}]}{\vdash [\Pi_{i=1}\ [\mathit{singleton}]]\ C_2\ [\Pi_{i=1}\ [\mathit{singleton}]]}\ \textsc{Specialize}}{\vdash \underbrace{[\Pi_{i=1}\ [\mathit{singleton}] \land (\forall\langle\varphi\rangle.\ \varphi^L(i) \in \{1,2\})]}_{P'}\ C_2\ \underbrace{[\Pi_{i=1}\ [\mathit{singleton}] \land (\forall\langle\varphi\rangle.\ \varphi^L(i) \in \{1,2\})]}_{Q'}}\ \textsc{FrameSafe} \tag{5.1}$$

$$\dfrac{\dfrac{\vdash [P]\ C_1\ [\mathit{hasMin}_x] \qquad \mathit{hasMin}_x \overset{\{i\}}{\Rightarrow} \mathit{mono}_x^i \land P' \qquad \dfrac{\dfrac{\vdash [\mathit{mono}_x^i]\ C_2\ [\mathit{mono}_y^i] \qquad \overset{(5.1)}{\vdash [P']\ C_2\ [Q']}}{\vdash [\mathit{mono}_x^i \land P']\ C_2\ [\mathit{mono}_y^i \land Q']}\ \textsc{And}}{\vdash [\mathit{mono}_x^i \land P']\ C_2\ [\mathit{hasMin}_y]}\ \textsc{Cons} \qquad \mathit{inv}^{\{i\}}(\mathit{hasMin}_y)}{\vdash [\mathit{hasMin}_x]\ C_2\ [\mathit{hasMin}_y]}\ \text{LU}}{\vdash [P]\ C_1; C_2\ [\mathit{hasMin}_y]}\ \textsc{Seq}$$

Figure 5.9.: A compositional proof that the sequential composition of a command that has a minimum and a monotonic, deterministic command in turn has a minimum. Recall that $\mathit{singleton} \triangleq (\exists\langle\varphi\rangle.\ \forall\langle\varphi'\rangle.\ \varphi = \varphi')$, and thus $\Pi_{i=1}\,[\mathit{singleton}] = (\exists\langle\varphi\rangle.\ \varphi(i) = 1 \land (\forall\langle\varphi'\rangle.\ \varphi'(i) = 1 \Rightarrow \varphi = \varphi'))$

V, written $P \overset{V}{\Rightarrow} P'$, iff

$$\forall S.\ P(S) \implies (\exists S'.\ P'(S') \land S \overset{V}{=} S')$$

*Finally, a hyper-assertion P is **invariant with respect to logical updates** in V, written $\mathit{inv}^V(P)$, iff*

$$\forall S_1, S_2.\ S_1 \overset{V}{=} S_2 \implies (P(S_1) \iff P(S_2))$$

Note that $\mathit{inv}^V(Q)$ means that Q cannot inspect the value of logical variables in V, but it usually also implies that Q cannot check for *equality* between states, and cannot inspect the cardinality of the set, since updating logical variables might collapse two states that were previously distinct (because of distinct values for logical variables in V).

Since this rule requires semantic reasoning, we also derive a weaker version of this rule, LUpdateS, which is easier to use. The rule LUpdateS allows us to strengthen a precondition P to $P \land (\forall\langle\varphi\rangle.\ \varphi(t) = e(\varphi))$, which corresponds to updating the logical variable t with the expression e, as long as the logical variable t does not appear *syntactically* in P, Q, and e (and thus does not influence their validity). For example, to connect the postcondition $\Box(x = 0 \lor x = 1)$ to the precondition $\mathit{mono}_x^t \triangleq (\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \varphi_1(t) = 1 \land \varphi_2(t) = 2 \Rightarrow \varphi_1(x) \geq \varphi_2(x))$ described in Section 5.2.2, one can use this rule to assign 1 to t if $x = 1$, and 2 otherwise. Appendix A.4.1 shows a detailed example.

5.7.2. Examples

We now illustrate our compositionality rules on two examples: Composing minimality and monotonicity, and composing non-interference with generalized non-interference.

Example 5.7.4 Composing minimality and monotonicity.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\vdash [\neg emp]\ C_2\ [\neg emp]}}{\vdash [\mathit{II}_{h=\varphi_1^L(h)}\ [\neg emp]]\ C_2\ [\mathit{II}_{h=\varphi_1^L(h)}\ [\neg emp]]}\ \textsc{Specialize}}{\vdash [low(l)]\ C_2\ [low(l)] \qquad \vdash [\exists\langle\varphi\rangle.\ \varphi^L(h) = \varphi_1^L(h)]\ C_2\ [\exists\langle\varphi\rangle.\ \varphi^L(h) = \varphi_1^L(h)]}\ \textsc{Cons}}{\vdash [low(l) \wedge (\exists\langle\varphi\rangle.\ \varphi^L(h) = \varphi_1^L(h))]\ C_2\ [low(l) \wedge (\exists\langle\varphi\rangle.\ \varphi^L(h) = \varphi_1^L(h))]}\ \textsc{And}}{\vdash \left[\bigotimes \left(low(l) \wedge (\exists\langle\varphi\rangle.\ \varphi^L(h) = \varphi_1^L(h))\right)\right]\ C_2\ \left[\bigotimes \left(low(l) \wedge (\exists\langle\varphi\rangle.\ \varphi^L(h) = \varphi_1^L(h))\right)\right]}\ \textsc{BigUnion}}{\vdash \underbrace{[\forall\langle\varphi_2\rangle.\ \exists\langle\varphi\rangle.\ \varphi_1^L(h) = \varphi^L(h) \wedge \varphi_2^P(l) = \varphi^P(l)]}_{P'_{\varphi_1}}\ C_2\ \underbrace{[\forall\langle\varphi_2\rangle.\ \exists\langle\varphi\rangle.\ \varphi_1^L(h) = \varphi^L(h) \wedge \varphi_2^P(l) = \varphi^P(l)]}_{Q'_{\varphi_1}}}\ \textsc{Cons} \tag{5.2}$$

$$\dfrac{\dfrac{\dfrac{\text{using (5.2) and } \varphi_1^L = \varphi_1'^L \implies Q'_{\varphi_1} = Q'_{\varphi_1'}}{\forall\varphi_1, \varphi_1'.\ (\varphi_1^L = \varphi_1'^L \wedge \vdash [\langle\varphi_1\rangle]\ C\ [\langle\varphi_1'\rangle] \implies (\vdash [P'_{\varphi_1}]\ C_2\ [Q'_{\varphi_1'}])}\ \textsc{Linking} \qquad \vdash [GNI_l^h]\ C_2\ [GNI_l^h]}{\vdash [low(l)]\ C_1\ [GNI_l^h] \qquad \vdash [low(l)]\ C_1; C_2\ [GNI_l^h]}\ \textsc{Seq}$$

Figure 5.10.: A compositional proof that the sequential composition of a command that satisfies GNI and a command that satisfies NI in turn satisfies GNI. Recall that $GNI_l^h \triangleq (\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \exists\langle\varphi\rangle.\ \varphi_1^L(h) = \varphi^L(h) \wedge \varphi^P(l) = \varphi_2^P(l))$, $low(l) \triangleq (\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \varphi_1^P(l) = \varphi_2^P(l))$, and $emp \triangleq (\forall\langle\varphi\rangle.\ \bot)$.

Consider a command C_1 that computes a function that has a minimum for x, and a deterministic command C_2 that is monotonic from x to y. We want to prove *compositionally* that $C_1; C_2$ has a minimum for y.

More precisely, we assume that the following hyper-triples are valid:

(1) $[P]\ C_1\ [hasMin_x]$ (minimality)
(2) $[mono_x^i]\ C_2\ [mono_y^i]$ (monotonicity)
(3) $[singleton]\ C_2\ [singleton]$ (determinism[14])

where $mono_x^i \triangleq (\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \varphi_1^L(i){=}1 \wedge \varphi_2^L(i){=}2 \implies \varphi_1^P(x) {\leq} \varphi_2^P(x))$, $hasMin_x \triangleq (\exists\langle\varphi\rangle.\ \forall\langle\varphi'\rangle.\ \varphi^P(x) \leq \varphi'^P(x))$, and $singleton \triangleq (\exists\langle\varphi\rangle.\ \forall\langle\varphi'\rangle.\ \varphi = \varphi')$, With the core rules alone, we cannot compose the two triples (1) and (2) to prove that $C_1; C_2$ has a minimum for y since the postcondition of C_1 does not imply the precondition of C_2.

Figure 5.9 shows a valid derivation in Hyper Hoare Logic of $\vdash [P]\ C_1; C_2\ [hasMin_y]$ (which we have proved in Isabelle/HOL). The key idea is to use the rule LUpdate to mark the minimal state with $i = 1$, and all the other states with $i = 2$, in order to match C_1's postcondition with C_2's precondition. Note that we *had to* use the consequence rule to turn C_2's postcondition $mono_y^i \wedge Q'$ into $hasMin_y$ *before* applying the rule LUpdate, because the latter hyper-assertion is invariant w.r.t. logical updates in $\{i\}$ (as required by the rule LUpdate), whereas the former is not.

The upper part of Figure 5.9 shows the derivation of $\vdash [P']\ C_2\ [Q']$, which uses the rule Specialize to restrict the triple $[singleton]\ C_2\ [singleton]$ to the subset of states where $i = 1$, ensuring the existence of a unique state (the minimum) where $i = 1$ after executing C_2. We also use the rule FrameSafe to ensure that our set only contains states with $i = 1$ or $i = 2$.[15]

14: This triple ensures that C_2 does not map the initial state with the minimum value for x to potentially different states with incomparable values for y (the order $\leq$ on values might be partial). Moreover, it ensures that C_2 does not drop any initial states because of an assume command or a non-terminating loop.

15: We could avoid the use of this rule by defining $mono_x^i$ as $\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \varphi_1^L(i){=}1 \wedge \varphi_2^L(i){\neq}2 \implies \varphi_1^P(x){\leq}\varphi_2^P(x)$.

Example 5.7.5 Composing non-interference with generalized non-interference.
To illustrate additional compositionality rules, we re-visit the example

introduced at the beginning of this section. Consider a command C_1 that satisfies GNI (for a public variable l and a secret variable h) and a command C_2 that satisfies NI (for the public variable l). We want to prove that $C_1; C_2$ satisfies GNI (for l and h).

More precisely, we assume that the following hyper-triples are valid:

(1) $\vdash [low(l)]\ C_1\ [GNI_l^h]$ (GNI)
(2) $\vdash [low(l)]\ C_2\ [low(l)]$ (NI)
(3) $\vdash [\neg emp]\ C_2\ [\neg emp]$ (termination)

where $GNI_l^h \triangleq (\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \exists\langle\varphi\rangle.\ \varphi_1^L(h) = \varphi^L(h) \wedge \varphi^P(l) = \varphi_2^P(l))$, $low(l) \triangleq (\forall\langle\varphi_1\rangle, \langle\varphi_2\rangle.\ \varphi_1^P(l) = \varphi_2^P(l))$, and $emp \triangleq (\forall\langle\varphi\rangle.\ \bot)$. Triple (3) is needed to ensure that C_2 does not drop executions depending on some values for h (*e.g.*, because of secret-dependent non-termination), which might cause $C_1; C_2$ to violate GNI.

Figure 5.10 shows a valid derivation of the triple $\vdash [low(l)]\ C_1; C_2\ [GNI_l^h]$ (which we have proved in Isabelle/HOL). The first key idea of this derivation is to use the rule Linking to eliminate the $\forall\langle\varphi_1\rangle$ in the pre- and postcondition of the triple $[GNI_l^h]\ C_2\ [GNI_l^h]$, while assuming that they have the same value for the logical variable h (implied by the assumption $\varphi_1^L = \varphi_1'^L$). The second key idea is to decompose any set of states S that satisfies P'_{φ_1} (defined as $\forall\langle\varphi_2\rangle.\ \exists\langle\varphi\rangle.\ \varphi_1^L(h) = \varphi^L(h) \wedge \varphi_2^P(l) = \varphi^P(l)$) into a union of smaller sets that all satisfy $low(l) \wedge (\exists\langle\varphi\rangle.\ \varphi_1^L(h) = \varphi^L(h))$. More precisely, we rewrite S as the union of all sets $\{\varphi, \varphi_2\}$ for all $\varphi, \varphi_2 \in S$ such that $\varphi_1^L(h) = \varphi^L(h) \wedge \varphi_2^P(l) = \varphi^P(l)$, using the rule Cons. Unlike S, these smaller sets all satisfy the precondition $low(l)$ of C_2, which allows us to leverage the triple $\vdash [low(l)]\ C_2\ [low(l)]$. Finally, we use the rule Specialize to prove that, after executing C_2 in each of the smaller sets $\{\varphi, \varphi_2\}$, there will exist at least one state φ' with $\varphi'^L(h) = \varphi_1^L(h)$.

5.8. Related Work

Overapproximate (relational) Hoare logics. *Hoare Logic* originated with the seminal works of Floyd [8] and Hoare [9], with the goal of proving programs functionally correct. *Relational Hoare Logic* [103] (RHL) extends Hoare Logic to reason about (2-safety) hyperproperties of a single program as well as *relational properties* relating the executions of two different programs (*e.g.*, semantic equivalence). RHL's ability to relate the executions of two different programs is also useful in the context of proving 2-safety hyperproperties of a single program, in particular, when the two executions take different branches of a conditional statement. In comparison, Hyper Hoare Logic can prove and disprove hyperproperties of a single program (Section 5.3.3), but requires a program transformation to express relational properties (see Appendix A.3.3). Extending Hyper Hoare Logic to multiple programs is interesting future work.

RHL has been extended in many ways, for example to deal with heap-manipulating [110] and higher-order [111] programs. A family of Hoare and separation logics [112–115] designed to prove non-interference [208] specialize RHL by considering triples with a single program, similar to Hyper Hoare Logic. Naumann [109] provides an overview of the

[8]: Floyd (1967), *Assigning Meanings to Programs*

[9]: Hoare (1969), *An Axiomatic Basis for Computer Programming*

[103]: Benton (2004), *Simple Relational Correctness Proofs for Static Analyses and Program Transformations*

[110]: Yang (2007), *Relational Separation Logic*

[111]: Aguirre et al. (2017), *A Relational Logic for Higher-Order Programs*

[112]: Amtoft et al. (2006), *A Logic for Information Flow in Object-Oriented Programs*

[113]: Costanzo et al. (2014), *A Separation Logic for Enforcing Declarative Information Flow Control Policies*

[114]: Ernst et al. (2019), *SecCSL*

[115]: Eilers et al. (2023), *CommCSL*

[208]: Volpano et al. (1996), *A Sound Type System for Secure Flow Analysis*

[109]: Naumann (2020), *Thirty-Seven Years of Relational Hoare Logic*

principles underlying relational Hoare logics. Sousa and Dillig [88] have proposed *Cartesian Hoare Logic* (CHL) to reason about k-safety properties for any fixed k, with a focus on automation and scalability. Hyper Hoare Logic can express the properties supported by CHL, in addition to many other properties; we show how to automate Hyper Hoare Logic in Chapter 6. D'Osualdo et al. [116] have identified several limitations of CHL when trying to compose together proofs of different k-safety properties, and have proposed a novel weakest-precondition calculus to overcome these limitations. Gladshtein et al. [117] have further extended the previous calculus to handle heap-manipulating programs and k-safety properties for values of k depending on program variables, which is useful to specify and verify computations over structured data (such as sparse representations of arrays or matrices).

[88]: Sousa et al. (2016), *Cartesian Hoare Logic for Verifying K-Safety Properties*

[116]: D'Osualdo et al. (2022), *Proving Hypersafety Compositionally*

[117]: Gladshtein et al. (2024), *Mechanised Hypersafety Proofs about Structured Data*

Underapproximate program logics. *Reverse Hoare Logic* [106] is an underapproximate variant of Hoare Logic, designed to prove the existence of good executions. The recent *Incorrectness Logic* [89] adapts this idea to prove the presence of bugs. Incorrectness Logic has been extended with concepts from separation logic to reason about heap-manipulating sequential [118] and concurrent [119] programs. It has also been extended to prove the presence of insecurity in a program (*i.e.*, to disprove 2-safety hyperproperties) [107]. Underapproximate logics [89, 106, 214] have been successfully used as foundation of industrial bug-finding tools [98–101]. Hyper Hoare Logic enables proving and disproving hyperproperties within the same logic.

[106]: de Vries et al. (2011), *Reverse Hoare Logic*

[89]: O'Hearn (2019), *Incorrectness Logic*

[118]: Raad et al. (2020), *Local Reasoning About the Presence of Bugs*

[119]: Raad et al. (2022), *Concurrent Incorrectness Separation Logic*

[107]: Murray (2020), *An Under-Approximate Relational Logic*

Several recent works have proposed approaches to unify over- and underapproximate reasoning. Exact Separation Logic [120] can establish both overapproximate and (backward) underapproximate properties over single executions of heap-manipulating programs, by employing triples that describe *exactly* the set of reachable states. Local Completeness Logic [217, 218] unifies over- and underapproximate reasoning in the context of abstract interpretation, by building on Incorrectness Logic, and enforcing a notion of *local completeness* (no false alarm should be produced relative to some fixed input). HL and IL have been both embedded in a Kleene algebra with diamond operators and countable joins of tests [219]. Dynamic Logic [220] is an extension of modal logic that can express both overapproximate and underapproximate guarantees over single executions of a program.

[89]: O'Hearn (2019), *Incorrectness Logic*
[106]: de Vries et al. (2011), *Reverse Hoare Logic*
[214]: Ascari et al. (2024), *Sufficient Incorrectness Logic*

[98]: Blackshear et al. (2018), *RacerD*
[99]: Gorogiannis et al. (2019), *A True Positives Theorem for a Static Race Detector*
[100]: Distefano et al. (2019), *Scaling Static Analyses at Facebook*
[101]: Le et al. (2022), *Finding Real Bugs in Big Programs with Incorrectness Logic*

[120]: Maksimović et al. (2023), *Exact Separation Logic*

[217]: Bruni et al. (2021), *A Logic for Locally Complete Abstract Interpretations*
[218]: Bruni et al. (2023), *A Correctness and Incorrectness Program Logic*

[219]: Möller et al. (2021), *On Algebra of Program Correctness and Incorrectness*

[220]: Harel et al. (1979), *First-Order Dynamic Logic*

Outcome Logic [102] (OL) unifies overapproximate and (forward) underapproximate reasoning for heap-manipulating and probabilistic programs, by combining and generalizing the standard overapproximate Hoare triples with forward underapproximate triples (see Appendix A.3.2). OL (instantiated to the powerset monad) uses a semantic model similar to our extended semantics (Definition 5.3.4), and a similar definition for triples (Definition 5.3.5). Moreover, a theorem similar to our Theorem 5.3.4 holds in OL, *i.e.*, invalid OL triples can be disproven within OL. The key difference with Hyper Hoare Logic is that OL does not support reasoning about hyperproperties. OL assertions are composed of atomic unary assertions, which can express the existence and the absence of certain states, but not relate states with each other, which is key to expressing hyperproperties. OL does not provide logical variables, on which we rely to express certain hyperproperties (see Section 5.2.2).

[102]: Zilberstein et al. (2023), *Outcome Logic*

Logics for $\forall^*\exists^*$-hyperproperties. Maillard et al. [104] present a general framework for defining relational program logics for arbitrary monadic effects (such as state, input-output, nondeterminism, and discrete probabilities), for two executions of two (potentially different) programs. Their key idea is to map *pairs* of (monadic) computations to relational specifications, using relational *effect observations*. In particular, they discuss instantiations for $\forall\forall$-, $\forall\exists$-, and $\exists\exists$-hyperproperties. RHLE [96] and FEHL [121] support overapproximate and (a limited form of) underapproximate reasoning, as they can establish $\forall^*\exists^*$-hyperproperties, such as generalized non-interference (Section 5.2.3) and program refinement. BiKAT [105], an algebra of alignment for relational verification, can be used directly to prove $\forall\forall$-properties. Moreover, $\forall\exists$-properties between two programs C_1 and C_2 can also be proved with BiKAT, by proving that a corresponding $\forall\forall$-property holds for some *alignment witness*, *i.e.*, a program that overapproximates the behavior of C_1 while underapproximating the behavior of C_2.[16]

[104]: Maillard et al. (2019), *The next 700 Relational Program Logics*

[96]: Dickerson et al. (2022), *RHLE*

[121]: Beutner (2024), *Automated Software Verification of Hyperliveness*

[105]: Antonopoulos et al. (2023), *An Algebra of Alignment for Relational Verification*

16: Note that one can in principle use BiKAT to prove $\exists\forall$-properties, by essentially proving the negation of $\forall\exists$-properties: To prove that an $\exists\forall$-property between two programs C_1 and C_2 holds, one needs to consider all programs W that overapproximate the behavior of C_1 and underapproximate the behavior of C_2, and prove that W does *not* satisfy a $\forall\forall$-property.

All four frameworks can be used to reason about relational properties of multiple programs, whereas Hyper Hoare Logic requires a program transformation to handle such properties. On the other hand, our logic supports a wider range of underapproximate reasoning and can express properties not handled by any of them, *e.g.*, $\exists^*\forall^*$-hyperproperties and hyperproperties relating an unbounded or infinite number of executions. Moreover, even for $\forall^*\exists^*$-hyperproperties, Hyper Hoare Logic provides while loop rules that have no equivalent in these logics, such as the rules *While*-$\exists$ (useful in this restricted context for $\exists^*$-hyperproperties) and *While*-$\forall^*\exists^*$ (Section 5.6). Additionally, as shown in Section 5.7, Hyper Hoare Logic offers flexible compositionality principles that are not supported by the aforementioned frameworks.

Probabilistic Hoare logics. Many assertion-based logics for probabilistic programs have been proposed [221–226]. These logics typically employ assertions over *probability (sub-)distributions* of states, which bear some similarities to hyper-assertions: Asserting the existence (resp. absence) of a state is analogous to asserting that the probability of this state is strictly positive (resp. zero). Taking the union of two sets of states is analogous to taking the sum of two sub-distributions. Our operator $\otimes$ (Definition 5.4.1) used in the rule *Choice* is thus similar to the operator $\oplus$ from Barthe et al. [224]. Notably, our loop rule *While*-$\forall^*\exists^*$ draws some inspiration from the rule *While* of Barthe et al. [224], which also requires an invariant that holds for all *unrollings* of the loop. These probabilistic logics have also been extended to the relational setting [227], for instance to reason about the equivalence of probabilistic programs.

[221]: Ramshaw (1979), *Formalizing the Analysis of Algorithms*
[222]: den Hartog (1999), *Verifying Probabilistic Programs Using a Hoare like Logic*
[223]: Corin et al. (2006), *A Probabilistic Hoare-style Logic for Game-Based Cryptographic Proofs*
[224]: Barthe et al. (2018), *An Assertion-Based Program Logic for Probabilistic Programs*
[225]: Barthe et al. (2019), *A Probabilistic Separation Logic*
[226]: Rand et al. (2015), *VPHL*

[224]: Barthe et al. (2018), *An Assertion-Based Program Logic for Probabilistic Programs*

[224]: Barthe et al. (2018), *An Assertion-Based Program Logic for Probabilistic Programs*

[227]: Barthe et al. (2009), *Formal Certification of Code-Based Cryptographic Proofs*

Verification of hyperproperties. The concept of hyperproperties has been formalized by Clarkson and Schneider [27]. Verifying that a program satisfies a k-safety hyperproperty can be reduced to verifying a safety property of the *self-composition* of the program [122, 216] (*e.g.*, by sequentially composing the program with renamed copies of itself). Self-composition has been generalized to product programs [123, 124]. (Extensions of) product programs have also been used to verify relational properties such as program refinement [125] and probabilistic relational properties such as differential privacy [228]. The temporal logics LTL,

[27]: Clarkson et al. (2008), *Hyperproperties*
[122]: Barthe et al. (2004), *Secure Information Flow by Self-Composition*
[216]: Terauchi et al. (2005), *Secure Information Flow as a Safety Problem*

[123]: Barthe et al. (2011), *Relational Verification Using Product Programs*
[124]: Eilers et al. (2019), *Modular Product Programs*

[125]: Barthe et al. (2013), *Beyond 2-Safety*

[228]: Barthe et al. (2014), *Proving Differential Privacy in Hoare Logic*

CTL, and CTL*, have been extended to HyperLTL and HyperCTL* [229] to specify hyperproperties, and model-checking algorithms [128, 230–232] have been proposed to verify hyperproperties expressed in these logics, including hyperproperties outside the safety class. Unno et al. [127] propose an approach to automate relational verification (including $\forall^*\exists^*$-properties such as GNI) based on an extension of constrained Horn-clauses. Relational properties of imperative programs can be verified by reducing them to validity problems in trace logic [233]. Finally, the notion of hypercollecting semantics [204] (similar to our extended semantics) has been proposed to statically analyze information flow using abstract interpretation [234]. One major difference between our extended semantics and this hypercollecting semantics is the treatment of loops. The former is defined directly on top of the big-step semantics (Definition 5.3.4), whereas the latter is defined inductively, and, in the case of loops, as a fixpoint over sets of sets of traces, which is more suitable for abstract interpretation, but less precise than the extended semantics. This difference in precision matters for hyperproperties that are not subset-closed (such as GNI) [235, 236].

[229]: Clarkson et al. (2014), *Temporal Logics for Hyperproperties*

[128]: Beutner et al. (2022), *Software Verification of Hyperproperties Beyond K-Safety*

[230]: Coenen et al. (2019), *Verifying Hyperliveness*

[231]: Hsu et al. (2021), *Bounded Model Checking for Hyperproperties*

[232]: Beutner et al. (2023), *AutoHyper*

[127]: Unno et al. (2021), *Constraint-Based Relational Verification*

[233]: Barthe et al. (2019), *Verifying Relational Properties Using Trace Logic*

[204]: Assaf et al. (2017), *Hypercollecting Semantics and Its Application to Static Analysis of Information Flow*

[234]: Cousot et al. (1977), *Abstract Interpretation*

[235]: Pasqua (2019), *Hyper Static Analysis of Programs – An Abstract Interpretation-Based Framework for Hyperproperties Verification*

[236]: Naumann et al. (2019), *Whither Specifications as Programs*

6. Hypra

Hyper! Hyper!

Scooter, *Hyper Hyper*

In the previous chapter, we have introduced Hyper Hoare Logic, a novel program logic to express and prove program hyperproperties with arbitrary quantifier alternations. In this chapter, we present Hypra, an automated deductive verifier for hyperproperties based on Hyper Hoare Logic, which inherits much of Hyper Hoare Logic's expressiveness, and thus is the first deductive verifier for hyperproperties with arbitrary quantifier alternations (including $\forall^*\exists^*$-hyperproperties and $\exists^*\forall^*$-hyperproperties).

6.1. Introduction

As we have seen in Chapter 1, automated deductive verifiers for hyperproperties are mostly limited to k-safety hyperproperties, which can be reduced to safety properties for a product program [122–124] and then verified using an off-the-shelf verifier. Alternatively, there are dedicated verifiers for hyperproperties, such as WhyRel [126], SecC (based on SecCSL [114]), and HyperViper (based on CommCSL [115]) for 2-safety hyperproperties, and Descartes (based on Cartesian Hoare Logic [88]) for k-safety hyperproperties.

To the best of our knowledge, ORHLE (based on RHLE) [96] and ForEx (based on FEHL) [121] are the only automated deductive verifiers that go beyond k-safety hyperproperties by supporting $\forall^*\exists^*$-hyperproperties.[1] However, they are limited to a *fixed* quantification scheme; users have to first fix the numbers of $\forall$-quantifiers and $\exists$-quantifiers and then write preconditions, postconditions, and loop invariants in this fixed scheme. It is, thus, not possible to compose proofs with different quantification schemes, *e.g.*, to use a $\forall$-property in the proof of a $\forall\forall$-property, or a $\forall\forall$-property in the proof of a $\forall\exists$-property. Moreover, ORHLE and ForEx only support *synchronized* loop rules,[2] which limits the programs and hyperproperties that can be verified in practice. For example, they cannot verify the monotonicity of the Fibonacci sequence (Example 5.6.3), which requires a loop rule such as Hyper Hoare Logic's While-$\forall^*\exists^*$.

To the best of our knowledge, no existing verifier supports properties beyond $\forall^*\exists^*$-hyperproperties, such as $\exists^*\forall^*$-hyperproperties.

This chapter. We present the first deductive verifier for hyperproperties with arbitrary quantifier alternations. Our tool, Hypra, is based on Hyper Hoare Logic (HHL) (Chapter 5). Like HHL, Hypra allows assertions to quantify explicitly over states, giving users the flexibility to express and combine different types of hyperproperties in the same proof. Going beyond HHL, Hypra supports reasoning about runtime errors (*e.g.*, to prove the existence or absence of bugs).

[122]: Barthe et al. (2004), *Secure Information Flow by Self-Composition*
[123]: Barthe et al. (2011), *Relational Verification Using Product Programs*
[124]: Eilers et al. (2019), *Modular Product Programs*
[126]: Nagasamudram et al. (2023), *The WhyRel Prototype for Modular Relational Verification of Pointer Programs*
[114]: Ernst et al. (2019), *SecCSL*
[115]: Eilers et al. (2023), *CommCSL*
[88]: Sousa et al. (2016), *Cartesian Hoare Logic for Verifying K-Safety Properties*
[96]: Dickerson et al. (2022), *RHLE*
[121]: Beutner (2024), *Automated Software Verification of Hyperliveness*

1: There are other approaches for automatically verifying $\forall^*\exists^*$-properties, such as asymmetric product programs [125], constrained Horn clauses [127], or predicate abstraction [128].

2: ORHLE's loop rule is similar to WhileSync, while ForEx's loop rule goes beyond this rule by supporting *fixed* alignments of loop iterations between different programs. For example, when verifying a relational property of the form $\forall\exists$ between two programs, one can choose to perform c_1 loop iterations in the first program and c_2 loop iterations in the second program, where c_1 and c_2 are fixed constants.

Our key insight is that arbitrary hyperproperties and the corresponding proof rules can be encoded into a standard intermediate verification language by representing *sets of states* of the input program *explicitly* in the states of the intermediate program. Verification is then automated using an existing SMT-based verifier for the intermediate language. To ensure that SMT solvers can handle the resulting verification conditions, our encoding carefully manages quantifier instantiations by tracking simultaneously a lower bound and an upper bound of the sets of states. Note that, in contrast to product constructions, our encoding does *not* duplicate the statements of the input program and can handle arbitrary hyperproperties (beyond k-safety).

Like HHL, we focus on hyperproperties that relate multiple executions of the *same* program; relating executions of *different* programs (*e.g.*, to prove their equivalence) poses additional challenges (such as finding an alignment), which are orthogonal to the problems addressed here. Both WhyRel and ORHLE support such relational proofs.

Contributions. In summary, this chapter makes the following contributions:

1. We extend Hyper Hoare Logic (Chapter 5) with the ability to reason about runtime errors. This allows us to prove correctness (the absence of bugs), incorrectness (the existence of bugs) and more complex hyperproperties (*e.g.*, proving that the occurrence of a runtime error does not leak any secret information).
2. We present the first approach to generate verification conditions for hyperproperties with arbitrary quantifier alternations for loop-free statements. The resulting verification conditions are amenable to SMT solvers.
3. HHL provides multiple loop rules for different kinds of programs and properties. We present our approach to automatically select which loop rule to apply, such that users are not exposed to the details of the underlying logic. This automatic selection is important when dealing with $\exists^*\forall^*$-invariants, which require the application of several different loop rules. Moreover, we present and prove sound a new loop rule for $\forall^*\exists^*$-hyperproperties, which is more suitable for automated verification than the corresponding rule in HHL.
4. We implement our approach in a tool called Hypra, based on the Viper intermediate language [16]. Our evaluation on a set of benchmarks from the literature shows that Hypra can prove a large class of hyperproperties for a large class of programs, in a reasonable amount of time and with a reasonable amount of proof annotations.

[16]: Müller et al. (2016), *Viper*

Outline. The rest of the chapter is organized as follows: Section 6.2 highlights the capabilities of our verifier through several examples, and presents our extension of Hyper Hoare Logic to reason about runtime errors. Section 6.3 presents our approach to generate verification conditions for loop-free statements, while Section 6.4 handles loops. Finally, we discuss the implementation and evaluation of Hypra in Section 6.5, and related work in Section 6.6.

Accompanying artifact. The artifact [137] associated with this chapter consists of (1) our tool Hypra, (2) our evaluation, with instructions on how to reproduce the results, and (3) Isabelle/HOL proofs of the soundness of the novel loop rule described in Section 6.4.1 (Theorem 6.4.1) and of Lemma 6.4.2.

[137]: Dardinier et al. (2024), *Hypra: A Deductive Program Verifier for Hyperproperties (Artifact)*

6.2. A Tour of the Verifier

In this section, we illustrate the key capabilities of Hypra on several examples. Section 6.2.1 shows how Hypra can be used to verify safety and reachability properties. Section 6.2.2 then illustrates how Hypra can verify complex hyperproperties (such as $\exists^*\forall^*$-hyperproperties). Section 6.2.3 illustrates how Hypra can be used to verify properties about runtime errors, such as the absence of errors, the existence of errors, and more complex (hyper)properties (such as the fact that the occurrence of a runtime error does not leak secret information). We also explain how we extend Hyper Hoare Logic, which does not support errors, to do so. Finally, Section 6.2.4 shows how Hypra handles while loops. All examples shown in this section are successfully and automatically verified by our tool.

6.2.1. Verifying Safety and Reachability Propertes

Hypra can be used to verify hyperproperties for the following class of programs:

Definition 6.2.1 Syntax of Hypra statements**.**
Hypra statements are written in the following syntax, where C ranges over program statements, x, x_i and y_i range over program variables, e ranges over arithmetic expressions, b ranges over boolean expressions, T over types, and m represents a method with n parameters and k return variables:

$$C ::= \textbf{skip} \mid \textbf{assume}\ b \mid \textbf{assert}\ b \mid C; C \mid x := \mathit{nonDet}() \mid x := e \mid \textbf{var}\ x : \mathbf{T}$$
$$\mid \textbf{if}\ (b)\ \{C\}\ \textbf{else}\ \{C\} \mid \textbf{while}\ (b)\ \{C\} \mid y_1, \ldots, y_k := m(x_1, \ldots, x_n)$$

This syntax is similar to the syntax from Definition 5.3.1. One addition is the statement **var** x : **T**, which declares a new local program variable x of type T. Another important addition is the statement $y_1, \ldots, y_k := m(x_1, \ldots, x_n)$, which calls the method m with arguments $x_1, \ldots, x_n$, and stores the results of the call in the variables $y_1, \ldots, y_k$. *Hypra programs* are collections of *methods*, which have a name, a list of typed parameters, a list of typed return variables, and an optional body (written in the syntax of Definition 6.2.1).

Specifications and hints. Given a method with body C, annotated with a precondition P and a postcondition Q, where P and Q are hyper-assertions written in the specification language that we will introduced in Section 6.2.3, Hypra tries to establish the hyper-triple $[P]\ C\ [Q]$. To help guide the proof, users can annotate while loops with loop invariants and

```
method randNat() returns (y: Int)
    requires ∃⟨σ⟩. ⊤
    ensures ∀⟨σ⟩. 1 ≤ σ(y) ≤ 2
    ensures ∃⟨σ⟩. σ(y) = 1
    ensures ∃⟨σ⟩. σ(y) = 2
{
    var x: Int
    x := nonDet() // {hint}
    // use hint(0,1)
    if (x > 0) {
        y := 1
    }
    else {
        y := 2
    }
}
```

```
method secure(h: Int, l: Int) returns (o: Int)
    requires ∀⟨σ1⟩,⟨σ2⟩. σ1(l) = σ2(l)
    ensures ∀⟨σ1⟩,⟨σ2⟩. σ1(o) = σ2(o)
{
    if (h > 0) { o := 2 * l }
    else { o := l }
    if (h <= 0) { o := o + l }
}
```

```
method leaky(h: Int) returns (o: Int)
    requires ∃⟨σ1⟩,⟨σ2⟩. σ1(h) < σ2(h)
    ensures ∃⟨σ1⟩,⟨σ2⟩. ∀⟨σ⟩. σ(o)=σ1(o)⇒σ(h)≠σ2(h)
{
    var y: Int
    y := nonDet() // {hint}
    assume 0 <= y <= 10
    // use hint(10)
    o := h + y
}
```

Figure 6.1.: Examples of overapproximation, underapproximation, and hyperproperties. The example on the left illustrates ∀-properties and ∃-properties of individual program executions. The examples on the right illustrate hyperproperties. Method `secure` satisfies non-interference, a ∀∀-hyperproperty. Method `leaky` (which is adapted from Example 5.2.7) violates generalized non-interference. The negation of this property is expressed as an ∃∃∀-hyperproperty. All examples are successfully verified by Hypra, using the provided hints to construct witnesses for existential quantifiers.

loop variants, as we will see in Section 6.2.4. Users can also provide *hints* for non-deterministic assignments, as we show in the next example.

Example 6.2.1 Verifying safety and reachability.
Consider the method `randNat` in Figure 6.1 (left). This method non-deterministically chooses a value for x (which can for example represent a random choice or some user input), and assigns 1 to y if $x > 0$, and 2 otherwise. In other words, this method non-deterministically returns 1 or 2. The keyword **`requires`** specifies the precondition of the method, while the keyword **`ensures`** specifies the postcondition. Multiple preconditions or postconditions are simply conjoined. Thus, verifying this example corresponds to proving the hyper-triple

$$[\exists\langle\sigma\rangle.\ \top]\ C_r\ [(\forall\langle\sigma\rangle.\ 1{\leq}\sigma(y){\leq}2) \wedge (\exists\langle\sigma\rangle.\ \sigma(y){=}1) \wedge (\exists\langle\sigma\rangle.\ \sigma(y){=}2)]$$

where C_r refers to the body of method `randNat`. Let S' be the set of reachable states at the end of the method. The postcondition $\forall\langle\sigma\rangle.\ 1 \leq \sigma(y) \leq 2$, which is equivalent to $\forall\sigma \in S'.\ 1 \leq \sigma(y) \leq 2$, *overapproximates* the set S': It means that, in any final state (from S'), y will either be 1 or 2, corresponding to the standard Hoare triple $\{\top\}\ C_r\ \{1 \leq y \leq 2\}$. On the other hand, the postconditions $\exists\langle\sigma\rangle.\ \sigma(y) = 1$ and $\exists\langle\sigma\rangle.\ \sigma(y) = 2$, equivalent to $\exists\sigma \in S'.\ \sigma(y) = 1$ and $\exists\sigma \in S'.\ \sigma(y) = 2$, respectively, *underapproximate* the set S': They express the existence of two reachable final states with $y = 1$ and $y = 2$. Conjoined, these three postconditions express that this method has only two possible outcomes for y, 1 and 2, and that both outcomes are reachable. The precondition $\exists\langle\sigma\rangle.\ \top$ expresses that the set of initial states is non-empty. This precondition is required for the hyper-triple to hold, otherwise the postconditions $\exists\langle\sigma\rangle.\ \sigma(y) = 1$ and $\exists\langle\sigma\rangle.\ \sigma(y) = 2$ would not hold (because no states are reachable from an empty set of initial states).

While Hypra verifies the first postcondition automatically, proving the existentially-quantified postconditions requires a user-provided hint. *Hints* are annotations for non-deterministic assignments that give examples of values that might be assigned. Hypra uses this information to construct witness states for ∃-properties. In our example, the `{hint}` annotation after the non-deterministic assignment introduces an identifier for this assignment, and the annotation **`use`** `hint(0,1)` tells the verifier that 0 and 1 are relevant choices for this non-deterministic assignment (technically, hints are used to provide triggers for the quantifier instantiation in the SMT solver).[3] The two values ensure that both branches of the subsequent conditional statement are considered, which is necessary to prove both existentially-quantified postconditions. The specific values are irrelevant in this example; any pair of a non-negative and a positive integer would work.

3: Note that, in general, hints can depend on the variables of one or more states. For example, **`use`** $\forall\langle\sigma\rangle.\,\texttt{hint}(\sigma(n))$ tells the SMT solver to consider the value of variable n in all relevant states σ. Hints can also depend on multiple quantified states, as in **`use`** $\forall\langle\sigma_1\rangle,\langle\sigma_2\rangle.\,\texttt{hint}(\sigma_1(a)+\sigma_2(b))$.

6.2.2. Verifying Hyperproperties

Overapproximation allows us to formally verify safety hyperproperties such as non-interference, as illustrated by the following example.

Example 6.2.2 Verifying non-interference.
Consider the method `secure` in Figure 6.1 (top right). The specification expresses that the output o depends only on the low-sensitive input l and, thus, does not leak any information about the secret input h. Hypra verifies this example without further annotations.

By enabling both overapproximation and underapproximation reasoning, our approach can verify more complex hyperproperties, such as $\forall^*\exists^*$-hyperproperties or $\exists^*\forall^*$-hyperproperties, as illustrated by the following example.

Example 6.2.3 Verifying a violation of generalized non-interference.
Consider the method `leaky` in Figure 6.1 (bottom right), inspired by Example 5.2.7. The statements `y := nonDet()` and **`assume`** `0 <= y <= 10` model a non-deterministic choice between 0 and 10. This method leaks information about its secret input h via its output o: h is between $o-10$ and o. To prove this claim, we formally verify that the method violates generalized non-interference [208, 210]. That is, we prove the existence of two executions with distinct secret values for h that can be *distinguished*. The postcondition expresses that observing the output $\sigma_1(o)$ rules out the possibility that the secret value of h was $\sigma_2(h)$, thus leaking information about the initial value of h. Verifying this existentially-quantified postcondition requires a hint; choosing the provided value 10 for $\sigma_2(y)$ yields the required witnesses.

[208]: Volpano et al. (1996), *A Sound Type System for Secure Flow Analysis*
[210]: McCullough (1987), *Specifications for Multi-Level Security and a Hook-Up*

6.2.3. Verifying Properties about Runtime Errors

Our examples so far reason about properties of *normal states*, that is, states that are reached when the program executes successfully. In addition, our technique can also reason about a set of *error states*, which are reached when a runtime error occurs. This feature allows us to prove

```
method buggy() returns (x: Int)
    requires ∃⟨σ⟩. ⊤
    ensures ∃⟨σ⟩_er. σ(x) = 1
    ensures ∃⟨σ⟩_er. σ(x) = 2
{
    x := randNat()
    var y: Int := x + x
    assert y % 2 == 1
}
```

```
method almostCorrect(x:Int) returns (o:Int)
    requires ∀⟨σ⟩. σ(x) ≥ 0
    ensures ∀⟨σ⟩_er. σ(x) = 0 ∧ σ(o) = 0
{
    o := nonDet()
    assume o >= 0
    var y: Int := x + o
    assert y > 0
}
```

```
method lowError(h: Int, l: Int, t: Int)
    requires ∃⟨σ1⟩,⟨σ2⟩. σ1(t) = 1 ∧ σ2(t) = 2
    requires ∀⟨σ1⟩,⟨σ2⟩. σ1(l) = σ2(l)
    ensures (∃⟨σ1⟩_er. σ1(t) = 1) ⇔ (∃⟨σ2⟩_er. σ2(t) = 2)
{
    if (h > 0) {
        assert l >= 0
    }
    if (l < 0) {
        assert false
    }
}
```

Figure 6.2.: Reasoning about runtime errors. In the top left example, the postconditions specify two possible executions that lead to a runtime error (an assertion violation). The postcondition on the top right expresses that the method fails *only if* both `x` and `o` are 0. The example on the bottom illustrates reasoning about runtime errors in the context of hyperproperties. The specification expresses that the occurrence of a runtime error does not depend on the secret input `h`. All examples are successfully verified by Hypra without any hints.

both the absence and presence of runtime errors, as well as advanced hyperproperties such as failure-sensitive non-interference.

Example 6.2.4 Verifying the existence of bugs**.**
Method `buggy` in Figure 6.2 (top left) calls method `randNat` (see Figure 6.1), doubles the result, and asserts that the resulting value is odd. The postconditions express the existence of two failing executions, à la Incorrectness Logic [89]: There exist error states σ where the result of `randNat` is 1 and 2, respectively.

[89]: O'Hearn (2019), *Incorrectness Logic*

The ability to quantify over error states allows us to express more complex properties, as illustrated by the following example.

Example 6.2.5 Verifying that a program is almost correct**.**
For example, the specification of method `almostCorrect` in Figure 6.2 (top right) expresses that a runtime error occurs *only if* the non-deterministic value assigned to o is 0 *and* the input x is 0. This *almost-correctness* is captured by the universal quantification in the postcondition which expresses that all error states satisfy $x = 0 \wedge o = 0$, that is, no other execution fails.

The absence of errors can be specified via the postcondition $\forall\langle\sigma\rangle_{\text{er}}.\ \bot$, which expresses that the set of error states is empty.

In the context of hyperproperties, reasoning about error states is, for instance, useful to express failure-sensitive non-interference, that is, the fact that observing a runtime error does not leak secret information, as shown by the following example.

Example 6.2.6 Verifying failure-sensitive non-interference.
The specification of method `lowError` in Figure 6.2 (bottom) expresses this property. For two different executions with the same value for the low-sensitive input l (but potentially different values for the secret input h), one execution fails if and only if the other execution fails. In particular, this proves that the occurrence of a runtime error does not depend on h, such that observing an error does not leak secret information. In this specification, the parameter t is used to *tag* executions, which allows us to identify the pre- and post-state of a given execution. Tags are expressed as logical variables in Hyper Hoare Logic, but represented by (immutable) program variables in Hypra.

Extending Hyper Hoare Logic to support runtime errors. We have extended Hyper Hoare Logic, which does not provide any support for reasoning about errors, as follows:

Definition 6.2.2 Hyper-triples with errors.
Given a program statement C and two program states σ and σ', we write $\langle C, \sigma\rangle \to \sigma'$ to express that executing C in the initial state σ may lead to the final normal state σ'. Executions that lead to runtime errors (because of violated assertions) are denoted as $\langle C, \sigma\rangle_{er} \to \sigma'$, where σ' is the last state reached before the error occurred. We refer to such states as ***error states****, in contrast to* normal states*. The set of* normal states *reachable by executing C in any state from S is denoted as $sem(C, S) \triangleq \{\sigma' \mid \exists \sigma \in S. \langle C, \sigma\rangle \to \sigma'\}$, while the set of* error states *reachable by executing C in any state from S is denoted as $err(C, S) \triangleq \{\sigma' \mid \exists \sigma \in S. \langle C, \sigma\rangle_{er} \to \sigma'\}$.*

Hyper-assertions *in Hypra are predicates over pairs of sets of states, where the first set corresponds to normal states, and the second set corresponds to error states. Given two hyper-assertions P and Q, we write $\models[P]\ C\ [Q]$ to express that the triple $[P]\ C\ [Q]$ is valid, which is defined as follows:*

$$\models[P]\ C\ [Q] \quad \textit{iff} \quad \forall S.\, P(S, \varnothing) \Rightarrow Q(sem(C, S), err(C, S))$$

Note that we start with an *empty* set of error states (second argument of P) to distinguish clearly between the errors caused by a statement C and those caused by statements executed prior to C. In particular, for a sequential composition $C_1; C_2$, the set of error states that come from C_2 depends on $sem(C_1, S)$ only, but not on $err(C_1, S)$; formally, $err(C_1; C_2, S) = err(C_1, S) \cup err(C_2, sem(C_1, S))$. Consequently, prescribing a specific set of error states in C_2's precondition is not useful.

We have also extended the syntax for hyper-assertions from Definition 5.3.3 to support quantification over error states, as follows.

Definition 6.2.3 Hypra's specification language.
Hypra supports the following syntax for hyper-assertions P where E ranges over integer expressions, B over Boolean expressions, and P over hyper-

```
method minimum(n: Int) returns (x: Int, y: Int)
    requires ∃⟨σ⟩. ⊤
    requires ∀⟨σ1⟩, ⟨σ2⟩. σ1(n) = σ2(n)
    ensures ∃⟨σ⟩. ∀⟨σ'⟩. σ(x) ≤ σ'(x) ∧ σ(y) ≤ σ'(y)
{
    var i, r: Int
    i, x, y := 0
    while (i < n)
        invariant ∀⟨σ1⟩, ⟨σ2⟩. σ1(n) = σ2(n) ∧ σ1(i) = σ2(i)
        invariant ∃⟨σ⟩. ∀⟨σ'⟩. σ(x) ≤ σ'(x) ∧ σ(y) ≤ σ'(y)
        decreases n - i
    {
        r := nonDet() // {hint}
        assume r >= 5
        // use hint(5)
        x := x + y + 2 * i + 3 * r
        y := x + 3 * i + 2 * r
        if (x >= n) { y := y + r }
        i := i + 1
    }
}
```

Figure 6.3.: Reasoning about loops. Given a loop invariant and an optional variant, Hypra automatically selects the appropriate loop rule. Like all other assertions, loop invariants may express arbitrary hyperproperties, here, the existence of an execution with minimal values for x and y. This example is successfully verified by Hypra.

assertions:

$$
\begin{aligned}
E &::= \sigma(y) \mid x \mid n \mid E + E \mid E - E \mid E * E \mid E \,/\, E \mid E \,\%\, E \mid \dots \\
B &::= \top \mid \bot \mid E = E \mid E > E \mid E \geq E \mid \neg B \mid \dots \\
P &::= \forall\langle\sigma\rangle.\, P \mid \exists\langle\sigma\rangle.\, P \mid \forall\langle\sigma\rangle_{er}.\, P \mid \exists\langle\sigma\rangle_{er}.\, P \mid \forall x.\, P \mid \exists x.\, P \\
&\quad \mid P \wedge P \mid P \vee P \mid P \Rightarrow P \mid B
\end{aligned}
$$

Hyper-assertions interact with the set of normal states through the quantifiers $\forall\langle\sigma\rangle$ and $\exists\langle\sigma\rangle$, and with the set of error states via the quantifiers $\forall\langle\sigma\rangle_{\text{er}}$ and $\exists\langle\sigma\rangle_{\text{er}}$. The quantifiers $\forall\langle\sigma\rangle_{\text{er}}$ and $\exists\langle\sigma\rangle_{\text{er}}$ are not allowed in preconditions (since we always start with an empty set of error states).

6.2.4. Verifying Loops

As discussed in Section 5.6, Hyper Hoare Logic provides four different loop rules that can prove different flavors of hyperproperties. These rules are applicable in different contexts; for example, some rules are applicable only if all loop executions perform the same number of iterations, and others only if the loop is proved to terminate. Based on a user-provided loop invariant (provided via the keyword **`invariant`**) and an optional loop variant (provided via the keyword **`decreases`**), Hypra determines automatically which rule to apply. This allows users to reason about loops in a familiar way without being exposed to the complexity of the underlying logic, as we illustrate in the following example.

Example 6.2.7 Verifying that a program has an execution with minimal values.
Consider the method `minimum` in Figure 6.3. This method starts with $x = y = 0$ and performs n loop iterations, during which it modifies the values of x and y in a non-deterministic way. We want to prove

that, given a fixed value for the input n (enforced by the precondition $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(n) = \sigma_2(n)$), there exists an execution where both x and y have minimal values at the end of the method (without specifying their values, which depend on n non-deterministic choices). The proof is based on a user-provided relational *loop invariant*, which must hold before the loop, and after every iteration. The first part of the loop invariant, $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\ \sigma_1(n) = \sigma_2(n) \land \sigma_1(i) = \sigma_2(i)$, ensures that all states have the same values for i and n, and thus that all executions will exit the loop simultaneously. Our verifier automatically detects this pattern, and uses a specialized loop encoding to handle it, as we will explain in Section 6.4. Moreover, knowing that all executions have the same value for i is necessary to prove the existence of an execution with minimal values. The second part of the loop invariant, $\exists\langle\sigma\rangle.\ \forall\langle\sigma'\rangle.\ \sigma(x) \leq \sigma'(x) \land \sigma(y) \leq \sigma'(y)$, ensures that, after any number of iterations, there exists an execution with minimal values for x and y, which corresponds to our postcondition. Finally, we need to prove that the loop terminates, otherwise our postcondition would not hold: If the loop did not terminate, then *no* final state would exist. We prove termination using a standard loop variant (following the **`decreases`** keyword). The verifier checks, for all states, that this loop variant is well-founded (non-negative), and that it strictly decreases during each iteration, thus ensuring termination.

This section illustrated the capabilities of our verification approach from a user's perspective. In the next two sections, we will explain how we compute verification conditions by encoding the input program into an intermediate verification language, for which an automated verifier exists.

6.3. Verification Conditions for Loop-Free Statements

Given a loop-free program statement C (potentially containing hints), a precondition P and a postcondition Q, our verifier generates a Viper [16] program, such that validity of the Viper program implies validity of the hyper-triple $\models[P]\ C\ [Q]$. Our key insight is to represent *sets of states* of the input program C as *single states* in the Viper program. More precisely, the Viper program contains set-valued variables, whose contents represent the set of states. Intuitively, the generated Viper program starts with a set-valued variable S (containing an arbitrary value), assumes that S satisfies the precondition P (via an assume-statement[4] in the Viper program), tracks the sets of normal states and error states that can be reached by executing C in any state from S (by updating the set-valued variable S accordingly), and checks whether they satisfy the postcondition Q (via an assert-statement[5] in the Viper program). To avoid clutter, we often ignore the set of error states in the rest of this chapter, and focus only on the set of normal states (which we also call the set of reachable states), but the same principles apply to both.

We first describe, in Section 6.3.1, a *simplified* version of our encoding. This simplified encoding is logically sound, but, as we show in Section 6.3.2, it does not work in practice, because it leads to matching loops where

[16]: Müller et al. (2016), *Viper*

4: Technically, we use an *inhale*-statement (Section 2.2.1), which has the same semantics as an assume statement for pure (non-SL) assertions.

5: Technically, we use an *exhale*-statement (Section 2.2.1), which has the same semantics as an assert statement for pure (non-SL) assertions.

the SMT solver gets stuck in an infinite instantiation of quantifiers. We then present our solution to this problem, which is to track *separately* an *upper bound* and a *lower bound* of the set of reachable states. Tracking both bounds separately avoids the aforementioned matching loops, but is sometimes too restrictive for preconditions that contain both universal and existential quantifiers. Thus, in Section 6.3.3, we show how we encode preconditions such that we can lift this restriction, while avoiding further matching loops.

6.3.1. (Naively) Tracking the Set of Reachable States

Our *simplified*[6] encoding starts with an arbitrary set of states S^0 that satisfies the precondition P. We then translates the Hypra command C to a Viper command $[\![C]\!]$, which computes from the set of initial states S^0 the set of final states $S^j \triangleq sem(C, S^0)$, for some $j \geq 0$.[7] Finally, we assert that the postcondition holds for the set of final states S^j. That is, for a triple $\models[P]\ C\ [Q]$, the Viper encoding generated by our approach has the following shape (for some j):

```
assume P(S⁰); [[C]]; assert Q(S^j)
```

6: We describe the actual encoding in Section 6.3.2.

7: In practice, S^0 and S^j correspond to the same variable whose value has been updated, but we use superscripts to denote the values of this variable at different points in the encoding, to simplify the explanations.

Encoding sequential compositions of atomic statements. Given an atomic statement C (assignments, assert statements, assume statements), our translation $[\![C]\!]$ to Viper, which updates the current set of states S^i to the set of reachable states $S^{i+1} \triangleq sem(C, S^i)$, is defined as follows (where $\langle C, \sigma_i\rangle \rightarrow \sigma_{i+1}$ is specialized for each atomic statement):

```
assume ∀σ_{i+1} ∈ S^{i+1}. ∃σ_i ∈ S^i. ⟨C, σ_i⟩ → σ_{i+1}
assume ∀σ_i ∈ S^i. ∀σ_{i+1}. ⟨C, σ_i⟩ → σ_{i+1} ⇒ σ_{i+1} ∈ S^{i+1}
```

The first line expresses that every state $\sigma_{i+1} \in S^{i+1}$ results from executing C in some state $\sigma_i \in S^i$, while the second line expresses that any final state σ_{i+1} that results from executing C in a state $\sigma_i \in S^i$ belongs to S^{i+1}. Together, the two assume statements ensure that $S^{i+1} = sem(C, S^i)$. Sequential compositions of commands are simply handled by chaining the two assume statements above, *i.e.*, $[\![C_1; C_2]\!] \triangleq [\![C_1]\!]; [\![C_2]\!]$.

Example 6.3.1 (Naively encoding a simple Hypra program)**.**
We illustrate our approach on the simple Hypra program `main` from Figure 6.4a. Figure 6.4b shows the high-level idea of our encoding into Viper (ignore the blue numbers for now).[8] We start with an arbitrary set of states S_0, which is assumed to satisfy the precondition P (Line 2). From S_0, we construct the set of reachable states $S_1 \triangleq sem(y := nonDet(), S_0)$ (Line 4 and Line 5). We then construct similarly the set $S_2 \triangleq sem(o := h + y, S_1)$ (Line 7 and Line 8). Finally, we assert that the postcondition Q holds for the set S_2 (Line 10).

8: As we will explain in Section 6.3.2, this encoding actually results in a *matching loop*, where the SMT solver gets stuck in an infinite instantiation of quantifiers. The actual Hypra encoding, which avoids this matching loop, is shown in Figure 6.4c.

Encoding conditional statements. For conditional statements, we leverage the fact that

$$\begin{aligned} &sem(\textbf{if } (b) \ \{C_1\} \textbf{ else } \{C_2\}, S) \\ =&sem(\textbf{assume } b; C_1, S) \cup sem(\textbf{assume } \neg b; C_2, S) \end{aligned}$$

Thus, to construct the encoding for a conditional statement, we split the current set of program states S^i into S^i_b and $S^i_{\neg b}$, where S^i_b is the set of states where b holds, and $S^i_{\neg b}$ is the set of states where b does not hold. S^i_b and $S^i_{\neg b}$ are then updated to S^j_b and $S^j_{\neg b}$, using $[\![C_1]\!]$ and $[\![C_2]\!]$, respectively. We finally compute the union of S^j_b and $S^j_{\neg b}$ to obtain the set of reachable states after the statement.

Tracking the set of error states. On top of tracking the set of reachable states, our encoding also tracks the set of error states $err(C, S)$ (defined in Definition 6.2.2) in the set-valued variable $S^\perp$. At the start of every method, the set of error states is initially empty (preconditions, unlike postconditions, are not allowed to quantify over error states). We then grow this set of error states monotonically, because the set of error states for a sequential composition $C_1; C_2$, written $err(C1; C2, S)$, is the union of $err(C_1, S)$ and $err(C_2, sem(C_1, S))$. Similarly, for conditionals, we compute the set of errors for both branches, and take their union. Error states can be generated only by `assert` statements: the encoding of **assert** b adds all reachable states that violate b to the set of error states. Finally, the error states arising from loops are handled via loop invariants (which we will see in Section 6.4). For example, to prove that there are no error states after a loop, the invariant must assert the absence of error states after every iteration.

E-matching and the encoding of hints. SMT solvers (such as Z3) used by Viper and other verifiers typically instantiate quantifiers via *E-matching* [237]. In this approach, every universal quantifier is associated with one or more syntactic matching patterns (also called *triggers*), ground terms that contain the bound variables of the quantifier. The quantifier gets instantiated *only* when the SMT solver's proof search encounters a term that matches its trigger (taking into account equalities). For instance, given the quantifier $\forall x. f(x) > 0$ with trigger $f(x)$, encountering the term $f(5)$ in the proof search will instantiate the quantifier with value 5 for the bound variable x.

[237]: Detlefs et al. (2005), *Simplify*

Triggers need to be chosen carefully. Overly restrictive triggers may prevent necessary quantifier instantiations, which may cause spurious verification errors. Conversely, overly permissive patterns may, in the worst case, introduce *matching loops*, where each quantifier instantiation produces a term that triggers the next instantiation, causing the SMT solver to diverge, as we will see next.

To avoid matching loops, our encoding uses rather restrictive triggers, as we discuss in Section 6.3.3. When our chosen triggers are too restrictive, users can initiate additional quantifier instantiations by annotating the input program with hints. These are encoded as applications of a vacuously-true function, which is used as a trigger. Consequently, a hint such as `hint(0,1)` in Figure 6.1 causes a quantifier instantiation with the

```
method main(l: Int) returns (y: Int, o: Int)
    requires ∀⟨σ⟩. ∃⟨σ'⟩. σ(l) ≠ σ'(l)
    ensures ∀⟨σ⟩. ∃⟨σ'⟩. σ(y) = σ'(y) ∧ σ(o) ≠ σ'(o)
{
    y := nonDet()
    o := l + y
}
```

(a) A simple Hypra program. This program would require a hint (*e.g.*, $\forall\langle\varphi\rangle.\,\text{hint}(\sigma(y))$) to be verified by Hypra, but we ignore this aspect for simplicity.

```
1  // translation of precondition:
2  assume ∀σ0 ∈ S0. ∃σ0' ∈ S0. σ(l) ≠ σ(l)
3  // translation of y := nonDet():
4  assume ∀(4,8) σ1 ∈ S1. ∃(5) σ0' ∈ S0. ∃v. σ1 = σ0[y := v]
5  assume ∀(6) σ0 ∈ S0. ∀v. ∃(7) σ1' ∈ S1. σ1 = σ0[y := v]
6  // translation o := h + y:
7  assume ∀(2) σ2 ∈ S2. ∃(3) σ1' ∈ S1. σ2 = σ1[o := σ1(h) + σ(y)]
8  assume ∀σ1 ∈ S1. ∃σ2' ∈ S2. σ2 = σ1[o := σ1(h) + σ(y)]
9  // translation of postcondition:
10 assert ∀(1) σ2 ∈ S2. ∃σ2' ∈ S2. σ(y) = σ(y) ∧ σ(o) ≠ σ(o)
```

(b) A naive encoding of the method `main` into Viper.

```
// translation of precondition:
assume ∀(6) σ0 ∈ S∀0. ∃(7) σ0' ∈ S∃0. σ(l) ≠ σ'(l)
// translation of y := nonDet():
assume ∀(4) σ1 ∈ S∀1. ∃(5) σ0' ∈ S∀0. ∃v. σ1 = σ0[y := v]
assume ∀(8) σ0' ∈ S∃0. ∀v. ∃(9) σ1' ∈ S∃1. σ1' = σ0'[y := v]
// translation of o := h + y:
assume ∀(2) σ2 ∈ S∀2. ∃(3) σ1' ∈ S∀1. σ2 = σ1[o := σ1(h) + σ(y)]
assume ∀(10) σ1' ∈ S∃1. ∃(11) σ2' ∈ S∃2. σ2' = σ1'[o := σ1'(h) + σ(y)]
// translation of postcondition:
assert ∀(1) σ2 ∈ S∀2. ∃(12) σ2' ∈ S∃2. σ(y) = σ'(y) ∧ σ(o) ≠ σ'(o)
```

(c) The actual Hypra encoding of the program `main` into Viper.

Figure 6.4.: A simple Hypra program (Figure 6.4a), and two potential encodings of this program into Viper. The naive encoding (Figure 6.4b) leads to a matching loop (4-5-6-7-8), while the encoding performed by Hypra (Figure 6.4c) avoids this problem.

values `0` and `1`. Hints are sometimes needed to instantiate the quantifier in the encoding of non-deterministic assignments; all other quantifiers are instantiated automatically.

6.3.2. Tracking Upper and Lower Bounds to Avoid Matching Loops

The encoding described above and shown in Figure 6.4b is logically sound, but it does not work in practice: Verification does not terminate because of *matching loops*, where the SMT solver gets stuck in an infinite instantiation of quantifiers, as we explain next.

Example 6.3.2 Matching loop in the simplified encoding.
Consider again the naive encoding of the method `main` shown in Figure 6.4b, where the blue numbers indicate the order in which the quantifiers are instantiated.[9]

1. The first quantifier to be instantiated is the universal quantifier 1 (Line 10), as the SMT solver tries to prove the postcondition, which gives us a state $\sigma_2 \in S_2$.
2. σ_2 then matches the universal quantifier 2 (Line 7),
3. which gives rise to a new state $\sigma_1' \in S_1$ (via the existential quantifier 3 on the same line).

9: We ignore some instantiations that do not matter for our example.

4. σ_1' matches the universal quantifier 4 (Line 4),
5. which gives rise to a new state $\sigma_0' \in S_0$ (existential quantifier 5).
6. σ_0' matches the universal quantifier 6 (Line 5),
7. which gives rise to a new state $\sigma_1'' \in S^1$ (existential quantifier 7).
8. σ_1'' matches the universal quantifier 4 (Line 4),
9. which gives rise to a new state $\sigma_0'' \in S^0$ (existential quantifier 5).
10. σ_0'' matches the universal quantifier 6 (Line 5),
11. which gives rise to a new state $\sigma_1''' \in S^1$ (existential quantifier 7).

And so on and so forth, resulting in an infinite cycle of quantifier instantiation.

To avoid matching loops, our key idea to track *separately* an *upper bound* $S^i_\exists$ and a *lower bound* $S^i_\forall$ of the set of reachable states.[10] For an atomic statement C, our translation $[\![C]\!]$ updates the current lower bound $S^i_\forall$ to the next lower bound $S^{i+1}_\forall$, and the current upper bound $S^i_\exists$ to the next upper bound $S^{i+1}_\exists$, as follows:[11]

```
assume ∀σ_{i+1}. σ_{i+1}∈S_∀^{i+1} ⇒ ∃σ_i. σ_i∈S_∀^i ∧ ⟨C, σ_i⟩→σ_{i+1}  // (LB)
assume ∀σ_i, σ_{i+1}. σ_i∈S_∃^i ∧ ⟨C, σ_i⟩→σ_{i+1} ⇒ σ_{i+1}∈S_∃^{i+1}  // (UB)
```

Starting with $S^i_\forall \subseteq S \subseteq S^i_\exists$, this encoding computes new values for $S^{i+1}_\forall$ and $S^{i+1}_\exists$ such that $S^{i+1}_\forall \subseteq \mathit{sem}(C, S) \subseteq S^{i+1}_\exists$, maintaining the invariant $S^i_\forall \subseteq \mathit{sem}(C, S) \subseteq S^i_\exists$ for all i. Note that the lower bound $S_\forall$ is sufficient to verify $\forall^*$-hyperproperties (safety hyperproperties), while the upper bound $S_\exists$ is sufficient to verify $\exists^*$-hyperproperties. Our tool uses this observation to optimize the encoding when only one kind of reasoning is needed, emitting only the encoding corresponding to $S_\forall$ (for $\forall^*$-hyperproperties) or $S_\exists$ (for $\exists^*$-hyperproperties). Both types of encoding are emitted when verifying both types of hyperproperties or hyperproperties with quantifier alternations.

Encoding preconditions and postconditions using $S_\forall$ and $S_\exists$. We apply the same encoding to preconditions and postconditions. We translate universally-quantified states (*i.e.*, $\forall\langle\sigma\rangle$) as universal quantifiers with range $S^i_\forall$ (*i.e.*, $\forall\sigma \in S^i_\forall$), and existentially-quantified states (*i.e.*, $\exists\langle\sigma\rangle$) as existential quantifiers with range $S^i_\exists$ (*i.e.*, $\exists\sigma \in S^i_\exists$). For preconditions, this encoding is necessary to establish $S^0_\forall \subseteq S^0 \subseteq S^0_\exists$, as universal state-quantifiers express *necessary* conditions for states to belong to S_0, whereas existential state-quantifiers express *sufficient* conditions for states to belong to S_0. For postconditions, we apply the same encoding because (LB) is useful when we have $\sigma_{i+1} \in S^{i+1}_\forall$ as an *assumption*, since we can then derive the existence of a state σ_i such that $\sigma_i \in S^i_\forall$ and $\langle C, \sigma_i\rangle \to \sigma_{i+1}$. Conversely, (UB) is useful when our goal is to *prove* $\sigma_{i+1} \in S^{i+1}_\exists$, since we can prove this goal by simply proving $\sigma_i \in S^i_\exists$ and $\langle C, \sigma_i\rangle \to \sigma_{i+1}$ for some state σ_i.

Example 6.3.3 Tracking upper and lower bounds separately avoids matching loops**.**
We show the encoding of the method `main` from Figure 6.4a based on upper and lower bounds in Figure 6.4c. The precondition (Line 2) and postcondition (Line 10) are encoded as explained above.

As in Example 6.3.2, the first quantifier to be instantiated is the

10: We also track lower and upper bounds of the set of error states, but we ignore this aspect here for simplicity.

11: In practice, we have four variables: $S_\forall$ and $S_\exists$, which represent the current bounds, and $S'_\forall$ and $S'_\exists$, which represent the next bounds, and our encoding is of the form `havoc S'; assume ...; S := S'`. We use the superscripts i and $i+1$ to denote the values of these variables at different points in the encoding, to simplify the explanations.

```
method simple(x,y:Int) returns (z:Int)
  requires ∀⟨σ1⟩. ∃⟨σ2⟩. σ2(x)>σ1(x)
  requires ∀⟨σ1⟩,⟨σ2⟩. σ1(y) = σ2(y)
  ensures ∀⟨σ1⟩. ∃⟨σ2⟩. σ2(z)>σ1(z)
{
    z := x + y
}
```

```
// next line is required to verify the program
assume S^0_∀ = S^0_∃
assume ∀σ1. σ1∈S^0_∀ ⇒ ∃σ2. σ2∈S^0_∃ ∧ σ2(x)>σ1(x)
assume ∀σ1,σ2. σ1∈S^0_∀ ∧ σ2∈S^0_∀ ⇒ σ1(y) = σ2(y)
⟦z := x + y⟧
assert ∀σ1. σ1∈S^1_∀ ⇒ ∃σ2. σ2∈S^1_∃ ∧ σ2(z)>σ1(z)
```

Figure 6.5.: A simple example that requires both overapproximation and underapproximation reasoning on the left, and its encoding on the right.

universal quantifier 1 from the postcondition on Line 10, which gives us a state $\sigma_2 \in S^2_\forall$. This state matches the universal quantifier 2 on Line 7, which gives us a state $\sigma'_1 \in S^1_\forall$. In turn, this state σ'_1 matches the universal quantifier 4 (Line 4), which gives us a state $\sigma'_0 \in S^0_\forall$ (existential quantifier 5). Crucially, this state σ'_0 *does not* match the universal quantifier 8 (Line 8), because it belongs to $S^0_\forall$, whereas the quantifier 8 requires a state from $S^0_\exists$, and thus the SMT solver avoids the matching loop discussed in Example 6.3.2.

At this point, the SMT solver can only instantiate the universal quantifier 6 (Line 2) with the state σ'_0, which produces a new state $\sigma''_0 \in S^0_\exists$ (via the existential quantifier 7). This state then successfully gives rise to a state $\sigma'_1 \in S^1_\exists$ (quantifiers 8 and 9 on Line 5),[12] and then to a state $\sigma'_2 \in S^2_\exists$ (quantifiers 10 and 11 on Line 8), which can be used as a witness for the existential quantifier (12) in the postcondition (Line 10).

12: For verification to succeed, the universally-quantified v on Line 5 can for example be instantiated with the value $\sigma_1(y)$.

Soundness. After $[\![C]\!]$, the sets $S^i_\forall$ and $S^i_\exists$ (and similarly for the lower and upper bounds of the set of error states) are *underspecified*; $[\![C]\!]$ ensures only that $S^i_\forall \subseteq \mathit{sem}(C,S) \subseteq S^i_\exists$. If the generated Viper program is successfully verified, then it is correct for all possible values of $S^i_\forall$ and $S^i_\exists$ after $[\![C]\!]$, provided that we started with a set S such that $P(S,\varnothing)$ holds. In particular, it is correct for $S^i_\forall = \mathit{sem}(C,S) = S^i_\exists$ (and similarly for the set of error states). Thus, successful verification of the generated Viper program implies that, for all S such that $P(S,\varnothing)$ holds, $Q(\mathit{sem}(C,S), \mathit{err}(C,S))$ holds, which corresponds exactly to the validity of the hyper-triple $\models[P]\ C\ [Q]$ (Definition 6.2.2).

6.3.3. Encoding Preconditions

Tracking separately an upper bound and a lower bound of the set of states as described in Section 6.3.2 avoids the matching loops described in Example 6.3.2. However, this encoding is too restrictive for some examples, where one wants to apply $\forall^*$-properties to existentially-quantified states. This requires the additional assumption that $S^0_\forall = S^0_\exists$, as we explain next.

Example 6.3.4 Applying $\forall^*$-properties to existentially-quantified states. Consider the method `simple` in Figure 6.5, with its encoding shown on the right. The precondition tells us that for any state σ_1, there exists a state σ_2 such that $\sigma_2(x) > \sigma_1(x)$, and that all states agree on the value of `y`. Thus, for any state σ_1, there should exist a state σ_2 such that

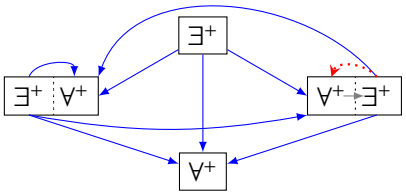

Figure 6.6.: Representation of the quantifier instantiations allowed by the chosen triggers.. Each node represents the shape of a possible part of the precondition; for simplicity, we omit shapes with more than one quantifier alternation here. The blue arrows show the existentially-quantified states that can be used to instantiate universal quantifiers, and the red arrow shows the instantiations our chosen triggers prevent. The grey arrow shows how instantiating the $\forall^+$-quantifiers in a $\forall^+\exists^+$-hyperproperty produces existentially-quantified states, which can subsequently be used to instantiate universal quantifiers. The acyclicity of the graph ensures the absence of matching loops among the involved quantifiers.

$\sigma_2(z) > \sigma_1(z)$ after the assignment. However, without the assumption that $S^0_\forall = S^0_\exists$ at the beginning of the method, this program would not be verified, as σ_2 belongs to $S^0_\exists$, but not necessarily to $S^0_\forall$, and thus one cannot prove that $\sigma_1(y) = \sigma_2(y)$. We explain next how we can equate the upper and lower bound for S^0 without introducing another matching loop.

Restricting quantifier instantiations for $\forall^*\exists^*$-preconditions. Assuming $S^0_\forall = S^0_\exists$ at the beginning of a method may lead to matching loops if the method precondition contains a $\forall^*\exists^*$-hyperproperty, as shown by the next example.

Example 6.3.5 A potential matching loop due to a $\forall^*\exists^*$-precondition. Consider the first precondition of the method in Figure 6.5, which is interpreted as $\forall\sigma_1 \in S^0_\forall.\, \exists\sigma_2 \in S^0_\exists.\, \sigma_2(x) > \sigma_1(x)$. The $\forall$-quantifier in the assertion introduces a new state $\sigma_2 \in S^0_\exists$ via the nested $\exists$-quantifier. Since we assume $S^0_\forall = S^0_\exists$, the new state σ_2 can trigger the instantiation of the same $\forall$-quantifier, which in turn can introduce a new state $\sigma_2' \in S^0_\exists$, and so on, leading to a matching loop.

Our solution is to use *limited* and *unlimited* functions [238] to control quantifier instantiations, to allow as many existentially-quantified states as possible to instantiate universal quantifiers, while avoiding matching loops. Concretely, when our tool translates a hyper-assertion with a universal state-quantifier such as $\forall\langle\sigma\rangle.\, P$ into Viper, it checks whether P contains an existential state-quantifier: If so, the $\forall$-quantifier is encoded with the *most restrictive* trigger, so that it can be instantiated only with those existentially-quantified states that do not occur under a universal quantifier. Otherwise, the $\forall$-quantifier is encoded with a more permissive trigger, allowing it to be instantiated by all existentially-quantified states.

[238]: Leino et al. (2009), *Reasoning about Comprehensions with First-Order SMT Solvers*

The effect of our solution is represented visually on Figure 6.6. Each node ($\exists^+$, $\exists^+\forall^+$, $\forall^+\exists^+$, $\forall^+$) represents the shape of a possible part of the precondition (the figure omits shapes with more than one quantifier alternation for simplicity). The blue arrows show the instantiations *enabled* by our tool, while the red dashed arrow shows the instantiation *prevented* by our tool via a restrictive trigger, since those instantiations can lead to matching loops. For example, states introduced by $\exists^+$-quantifiers can be used to instantiate the $\forall^+$-quantifiers of a $\forall^+\exists^+$-hyperproperty, which can

in turn be used to instantiate the $\forall^+$-quantifiers of a $\forall^+$-hyperproperty. As a more concrete example, to verify the method `simple` from Figure 6.5, the state σ_2 coming from the existential quantifier of the first precondition $\forall\langle\sigma_1\rangle.\, \exists\langle\sigma_2\rangle.\, \sigma_2(x) > \sigma_1(x)$ can be used to instantiate a $\forall$-quantifier of the second precondition $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(y) = \sigma_2(y)$, since there is a blue arrow from the right of the node $\forall^+\exists^+$ to the node $\forall^+$.

Crucially, states introduced by existential quantifiers that are nested under universal quantifiers *cannot* be used to instantiate the universal quantifiers in $\forall^+\exists^+$-hyperproperties (as represented by the red dotted arrow), since this would create a cycle and thus lead to a matching loop, as illustrated above with the first precondition of the method `simple`. As can be seen in Figure 6.6, preventing this instantiation makes the graph acyclic, which ensures the absence of matching loops.

6.4. Verification Conditions for Loops

In the previous section, we described the verification conditions generated by our verifier for loop-free statements. In this section, we describe how to generate verification conditions for while loops. We first describe, in Section 6.4.1, the different rules offered by Hyper Hoare Logic to reason about while loops, how we can derive verification conditions from them, and how our verifier automatically selects the right rule(s) to apply, based on the context. In Section 6.4.2, we discuss one such particular rule, the rule While-$\forall^*\exists^*$ and show that it cannot be used directly for our purpose; a *naive* encoding based on this rule would be unsound. We then present and prove sound (in Isabelle/HOL [33]) a novel loop rule suitable for automated deductive verification, WhileAuto-$\forall^*\exists^*$, which can be used in the same context. Finally, in Section 6.4.3, we present a technique to automatically frame information around the loop, which overcomes a limitation of these loop rules, and leads to more succinct loop invariants.

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

6.4.1. Automatically Generating Verification Conditions

Reasoning about loops in a relational setting is notoriously hard. In the context of deductive verification, our goal is to automatically reason about while loops, while keeping the amount of proof annotations needed from the user to a minimum. As illustrated in Figure 6.3, this means that the user should only provide a loop invariant, and optionally a loop variant (**`decreases`** clause).

Figure 6.7 shows the four rules leveraged by Hypra to verify while loops. The rules WhileSync and WhileSyncTot are the same as described in Section 5.6, where the latter uses *terminating hyper-triples* $\models_{\Downarrow} [P]\ C\ [Q]$ (formally defined at the end of Section 5.3.2). Terminating hyper-triples are stronger than normal hyper-triples, in that they additionally require the existence of a terminating execution from any initial state. In Hypra, we ensure this requirement by proving that all loops in C terminate, through the use of well-founded loop variants.[13] The third rule, WhileAuto-$\exists$, is adapted from the rule While-$\exists$ presented in Section 5.6.3, as we'll

13: In theory, we also need to check that **`assume`** statements do not break this property. By default, Hypra leaves this responsibility to the user, since **`assume`** statements are typically used to restrict non-deterministic assignments to the right range (as done in the example from Figure 6.3), which does not break this property. However, Hypra provides the more conservative option, disabled by default, to check the absence of **`assume`** statements within statements whose termination is required.

$$\textsc{WhileSync}\quad \frac{I \models \mathit{low}(b) \qquad \models [I \wedge \Box b]\ C\ [I]}{\models [I]\ \textbf{while}\ (b)\ \{C\}\ [(I \vee \Box\bot) \wedge \Box(\neg b)]}$$

$$\textsc{WhileSyncTot}\quad \frac{I \models \mathit{low}(b) \qquad \models_{\Downarrow} [I \wedge \Box(b \wedge e = t)]\ C\ [I \wedge \Box(e \prec t)] \qquad t \notin \mathit{fv}(I) \cup \mathit{mod}(C)}{\models_{\Downarrow} [I]\ \textbf{while}\ (b)\ \{C\}\ [I \wedge \Box(\neg b)]}$$

$$\textsc{WhileAuto-}\forall^*\exists^*\quad \frac{\models [I]\ \textbf{if}\ (b)\ \{C\}\ [I] \qquad \text{no } \forall\langle\_\rangle \text{ after any } \exists \text{ in } I}{\models [I]\ \textbf{while}\ (b)\ \{C\}\ [\Theta_{\neg b}(I) \wedge \Box(\neg b)]}$$

$$\textsc{WhileAuto-}\exists\quad \frac{\forall v.\ \models [(\exists\langle\sigma\rangle.\ P_\sigma \wedge b(\sigma) \wedge v = e(\sigma)) \wedge R]\ \textbf{if}\ (b)\ \{C\}\ [(\exists\langle\sigma\rangle.\ P_\sigma \wedge e(\sigma) \prec v) \wedge R] \qquad \forall\sigma.\ \models [P_\sigma \wedge R]\ \textbf{while}\ (b)\ \{C\}\ [P_\sigma \wedge R]}{\models [(\exists\langle\sigma\rangle.\ P_\sigma) \wedge R]\ \textbf{while}\ (b)\ \{C\}\ [(\exists\langle\sigma\rangle.\ P_\sigma) \wedge R \wedge \Box(\neg b)]}$$

Figure 6.7.: The rules applied by Hypra to reason about while loops. In the rules WhileSyncTot and WhileAuto-∃, the order $\prec$ must be *well-founded*. Moreover, $\mathit{low}(b) \triangleq (\forall\langle\sigma\rangle, \langle\sigma'\rangle.\ b(\sigma) = b(\sigma'))$ and $\Box b \triangleq (\forall\langle\sigma\rangle.\ b(\sigma))$. Finally, $\models_{\Downarrow} [P]\ C\ [Q]$ corresponds to a *terminating* hyper-triple. Terminating hyper-triples (defined at the end of Section 5.3.2) are stronger than normal hyper-triples, in that they additionally require the existence of a terminating execution from any initial state.

discuss below. Finally, the rule WhileAuto-$\forall^*\exists^*$ is a novel rule for $\forall^*\exists^*$-hyperproperties, which we discuss in Section 6.4.2.

As shown by Figure 6.7, all four rules use a loop invariant: I in the first three loop rules, and $\exists\langle\sigma\rangle.\ P_\sigma$ in the last rule. Moreover, those rules are non-obvious, which makes it hard for users to know which rule to apply in which context.

In the following, we explain the role of the different rules, how we derive verification conditions from them, and how our verifier automatically chooses the relevant rule(s), based on the user-provided loop invariant and optional loop variant. The Viper encodings of loops based on these rules are shown in Figure 6.8.

Synchronized loop rules. The two first rules, WhileSync and WhileSyncTot, apply when all executions exit the loop simultaneously. The key difference between the two rules can be seen in the postconditions of their conclusions: On top of the fact that all states satisfy the negation of the loop guard ($\Box(\neg b)$), the rule WhileSyncTot allows us to assume that the relational invariant I holds after the loop, whereas the rule WhileSync allows us to assume only that $I \vee \Box\bot$ holds after the loop, which is weaker than I. The $\Box\bot$ disjunct, which corresponds to the case where the loop does not terminate, is problematic when we want to prove postconditions with top-level existentially-quantified states. In this case, we need to use the rule WhileSyncTot, which requires us to prove that the loop terminates. The latter can be achieved by proving that a well-founded variant e strictly decreases after every iteration.

Verification conditions can be easily derived from these two rules, as shown in Figure 6.8a and Figure 6.8b, respectively. First, for both rules, we check that the user-provided loop invariant I entails $\mathit{low}(b)$. Then, for the rule WhileSync, we separately check the triple $\models [I \wedge \Box b]\ C\ [I \vee \Box\bot]$, as described in Section 6.3. For the rule WhileSyncTot, we instantiate e with the user-provided loop variant (required to be an integer expression), and separately check the triple $\models_{\Downarrow} [I \wedge \Box(b \wedge e = t)]\ C\ [I \wedge \Box(0 \leq e < t)]$, where t is a fresh variable. The check $0 \leq e$ ensures that the user-provided variant is well-founded. To ensure that this triple is a terminating hyper-triple, we check (syntactically) that all loops within C are annotated with **`decreases`** clauses, which ensures termination (provided that verification is successful). Finally, for both rules, we assert that the loop invariant I

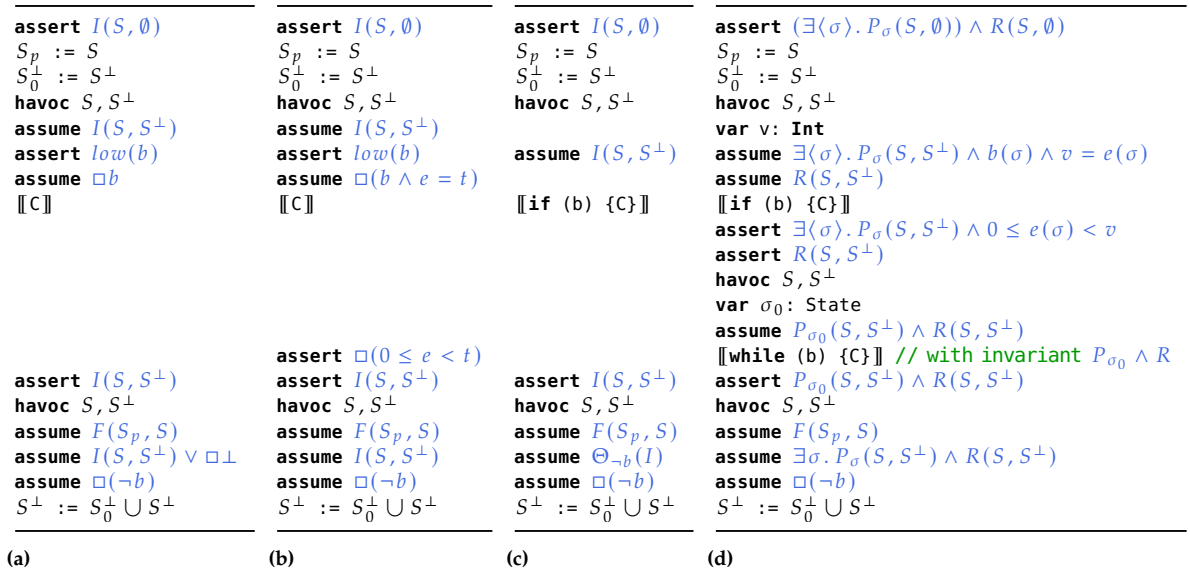

Figure 6.8.: Viper encodings of loops based on (a) the rule WhileSync, (b) the rule WhileSyncTot, (c) the novel rule WhileAuto-$\forall^*\exists^*$ for $\forall^*\exists^*$-hyperproperties and (d) the rule WhileAuto-$\exists$. To avoid clutter, we use S to represent both $S_\forall$ and $S_\exists$, and $S^\perp$ to represent both $S_\forall^\perp$ and $S_\exists^\perp$. We also use the notation ⟦C⟧ to refer to the encoding of the command C. In the loop encodings, b is the loop guard, C is the loop body, v and t are fresh variables, S_p is an auxiliary variable recording the value of the set of states before the loop (see Section 6.4.3), e is the loop variant, I and P_σ are loop invariants encoded as a predicate dependent on S and $S^\perp$. Moreover, $low(b) \triangleq (\forall\langle\sigma\rangle,\langle\sigma'\rangle.\, b(\sigma) = b(\sigma'))$, $\Box(b) \triangleq (\forall\langle\sigma\rangle.\, b(\sigma))$, and $F(S_p,S)$ corresponds to automatic framing as described in Section 6.4.3.

holds before the loop, and assume that $(I \vee \Box\bot) \wedge \Box(\neg b)$ (rule WhileSync) or $I \wedge \Box(\neg b)$ (rule WhileSyncTot) holds after the loop.

Non-synchronized loop rules. The two remaining loop rules from Figure 6.7, WhileAuto-$\forall^*\exists^*$ and WhileAuto-$\exists$, can be applied when different executions exit the loop at potentially different times. In this case, our premises are more complex: We need to reason about the *unrollings* of the while loop, which we achieve by proving a loop invariant over **if** (b) $\{C\}$ (in contrast to C for the synchronized rules). The novel rule WhileAuto-$\forall^*\exists^*$, which we present in Section 6.4.2, is more suitable for automated verification than the corresponding rule While-$\forall^*\exists^*$ from Figure 5.5, as it requires only one user-provided invariant I, whereas the latter additionally requires a user-provided postcondition Q. We show the encoding based on this rule in Figure 6.8c.

Similarly, the rule WhileAuto-$\exists$ is adapted from the rule While-$\exists$ presented in Section 5.6.3, but it differs from the latter in two ways. First, it requires only one invariant, whereas the rule While-$\exists$ additionally requires a (potentially different) postcondition $\exists\langle\sigma\rangle.\, Q_\sigma$. Second, while the rule While-$\exists$ requires the invariant to be of the form $\exists\langle\sigma\rangle.\, P_\sigma$, the rule WhileAuto-$\exists$ requires the more flexible form $(\exists\langle\sigma\rangle.\, P_\sigma) \wedge R$.[14]

The encoding based on the rule WhileAuto-$\exists$ is shown in Figure 6.8d. As before, Hypra specializes the well-founded order $\prec$ to be the canonical well-founded order over natural numbers. Moreover, note that, in both premises, v and σ are *meta-variables*, *i.e.*, there is not one value of v (in the first premise) or σ (in the second premise) per state, but rather there is one per *set* of states. In practice, to check the first premise, we use a fresh unconstrained variable v, and assume that the set of states and the variable v together satisfy the precondition $(\exists\langle\sigma\rangle.\, P_\sigma \wedge b(\sigma) \wedge v = e(\sigma)) \wedge R$, and

14: To apply this rule in practice, Hypra checks whether any line of the user-provided invariant has a top-level existential quantifier, and if so, treats the conjunction of all other lines as R. If no line has a top-level existential quantifier, and no other rule is applicable, Hypra emits an error message to inform the user that the program cannot be verified.

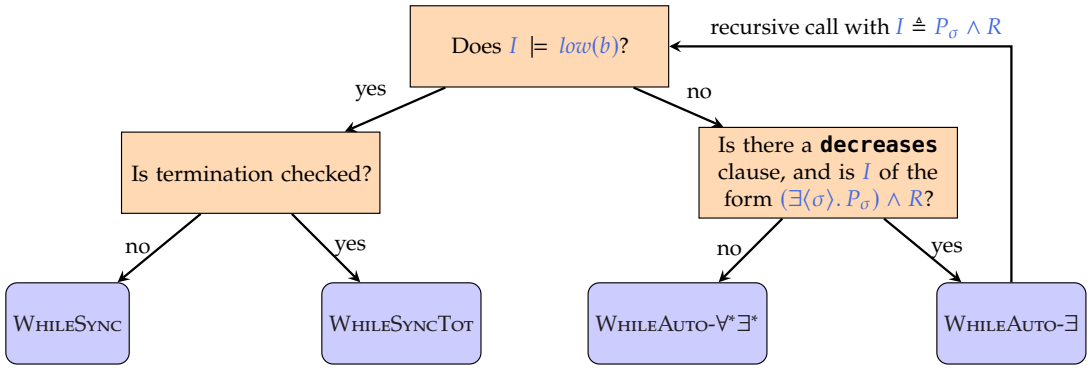

Figure 6.9.: Automatic loop rule selection to verify a loop **while** (b) $\{C\}$ with the invariant I. The first choice checks whether all executions perform the same number of loop iterations. The next choice is based on a subtle aspect of the loop's termination: The branch on the right checks whether the given loop performs a finite number of iterations, but does *not* require nested loops to terminate; this is sufficient to apply WhileAuto-∃. In contrast, the branch on the left requires the loop and all nested loops to terminate, as required by the rule WhileSyncTot. While the top-level choice is made semantically by the encoding, the next-level choices are determined syntactically based on the presence of **decreases** clauses. The edge from WhileAuto-∃ back to the first choice reflects the second premise of this rule, $\forall\sigma.\models[P_\sigma \wedge R]$ **while** (b) $\{C\}$ $[P_\sigma \wedge R]$: To check that this premise holds, Hypra recursively calls the rule selection procedure, which will automatically select a new loop rule adapted to the new loop invariant $P_\sigma \wedge R$.

check (after the encoding of **if** (b) $\{C\}$) that the set of states and the variable v together satisfy the postcondition $(\exists\langle\sigma\rangle.\, P_\sigma \wedge 0 \leq e(\sigma) < v) \wedge R$. Checking the second premise is more complicated, since it requires reasoning about the same while loop. However, note that the precondition (and postcondition) of this premise, $P_\sigma \wedge R$, is syntactically smaller than the precondition (and postcondition) of the conclusion of the rule, $(\exists\langle\sigma\rangle.\, P_\sigma) \wedge R$. Hypra automatically generates the verification conditions for this premise, using the approach described in this section, by automatically selecting the right loop rule based on the new loop invariant $P_\sigma \wedge R$ and the same loop variant e.

Automatically selecting the right loop rule(s). As explained at the start of this section, using only the user-provided loop invariant I and optional loop variant e, Hypra automatically selects the right loop rule(s) to apply, as depicted in Figure 6.9. First, we check (semantically in our encoding) whether the loop invariant guarantees that all executions will exit the loop simultaneously, that is, whether $I \models low(b)$ holds. If so, we apply one of the two synchronized loop rules, WhileSync or WhileSyncTot, depending on whether the user provided a loop variant for this loop and all loops nested within. Compared to non-synchronized loop rules, these two rules, when applicable, have weaker premises (i.e., their premises can be derived from the premises of the non-synchronized loop rules) and stronger conclusions (e.g., their postconditions after the while loop are at least as strong as those of the non-synchronized loop rules). Therefore, the synchronized loop rules are always better than the non-synchronized loop rules when they are applicable. The rule WhileSyncTot is the most powerful (when it applies), because its premise only requires reasoning about C, which is easier than reasoning about **if** (b) $\{C\}$, and the postcondition of its conclusion, $I \wedge \Box(\neg b)$, is not weaker than the postcondition given by any other rule (Section 6.4.2 will make clearer why the postcondition of the rule WhileAuto-∀*∃* is weaker). When termination is not checked (*i.e.*, no loop variant is provided), the choice is only between the rules WhileSync and WhileAuto-∀*∃*, and Hypra applies WhileSync whenever possible, for similar reasons.

When $I \not\models \mathit{low}(b)$, we apply one of the two non-synchronized loop rules, WhileAuto-$\forall^*\exists^*$ or WhileAuto-$\exists$, depending on the shape of the loop invariant I. When I is of the form $\forall^+\exists^*$, we can apply only the rule WhileAuto-$\forall^*\exists^*$, because our invariant is not of the form $\exists\langle\sigma\rangle.\, P \wedge R$, required by the rule WhileAuto-$\exists$. Similarly, when I is of the form $\exists^+\forall^+$, we can apply only the rule WhileAuto-$\exists$, because I does not satisfy the syntactic restriction from the rule WhileAuto-$\forall^*\exists^*$. Thus, there exists a *choice* between those two loop rules only when I has the shape $\exists^+$ and a loop variant is provided (otherwise the rule WhileAuto-$\exists$ cannot be applied). In this case, the rule WhileAuto-$\exists$ is more powerful, since it easily allows proving that the existentially-quantified states in I will still exist after the while loop, thanks to the loop variant, as illustrated by the following example.

Example 6.4.1 Using the rule WhileAuto-$\exists$ for $\exists^+$-invariants.
Let $I \triangleq (\exists\langle\sigma\rangle.\, \sigma(x) = \sigma(y))$. We can use the rule WhileAuto-$\exists$ with $P_\sigma \triangleq (\sigma(x) = \sigma(y))$ (and $R \triangleq \top$), which gives us the desired postcondition $\exists\langle\sigma\rangle.\, \sigma(x) = \sigma(y)$ in its conclusion.

In contrast, the postcondition of the conclusion of the rule While-$\forall^*\exists^*$ (on which the rule WhileAuto-$\forall^*\exists^*$ is based) is some hyper-assertion Q, such that $\models[\exists\langle\sigma\rangle.\, \sigma(x) = \sigma(y)]$ **assume** $\neg b$ $[Q]$ holds. In particular, we can get our desired postcondition $\exists\langle\sigma\rangle.\, \sigma(x) = \sigma(y)$ only if $x = y$ (in a state σ) implies $\neg b$, which will typically not be the case (because $\exists\langle\sigma\rangle.\, \sigma(x) = \sigma(y)$ is our loop invariant, which has to already hold *before* the loop).

Thus, when applicable, Hypra applies the rule WhileAuto-$\exists$ over the rule WhileAuto-$\forall^*\exists^*$, which then recursively applies the same automatic loop rule selection with the smaller loop invariant P, as shown in Figure 6.9.

6.4.2. $\forall^*\exists^*$-Hyperproperties

When I is of the form $\forall^+\exists^*$, and does not imply $\mathit{low}(b)$, the only loop rule from Section 5.6 that can be applied is the rule While-$\forall^*\exists^*$. Automating this rule is surprisingly not straightforward, as we explain next, which is why we introduce the novel rule WhileAuto-$\forall^*\exists^*$. Checking the first premise $\models[I]$ **if** (b) $\{C\}$ $[I]$ of the rule While-$\forall^*\exists^*$ is easy, as it can be checked separately using the user-provided loop invariant I. However, deriving from the loop invariant I a suitable postcondition Q that satisfies the syntactic restriction is more challenging. In the following, we first show why the naive *semantic* approach for deriving Q does not work, and then discuss our solution, which derives Q *syntactically* from I.

Naively deriving Q semantically is unsound. A natural idea is to check the syntactic restriction (no universal state-quantifier should occur under an existential quantifier) on I instead of Q, and then to derive Q from I *semantically*, *i.e.*, we can obtain the postcondition Q implicitly by considering a fresh set of states after the loop, assuming that it satisfies I, and then encoding **assume** $\neg b$. We cannot check the syntactic restriction on Q (as mandated by the rule) directly, since Q is not a *syntactic* hyper-assertion. In essence, this naive encoding corresponds to the following

```
method naive_encoding(t: Int, n: Int) returns (x: Int)
    requires ∃⟨σ1⟩. σ1(t) = 1
    requires ∀v. v ≥ 0 ⇒ ∃⟨σ2⟩. σ2(t) = 2 ∧ σ2(n) = v
    ensures ∃v. ∀⟨σ⟩. σ(x) ≤ v // this postcondition does not hold
{
    x := 0
    while (t = 1 ∨ x < n)
        invariant ∃⟨σ1⟩. σ1(t) = 1
        invariant ∀⟨σ1⟩,⟨σ2⟩. σ1(t) = 1 ⇒ σ1(x) ≥ σ2(x)
    {
        x := x + 1
    }
}
```

Figure 6.10.: An example showing why a *naive* encoding based on the rule WHILE-∀*∃* would be unsound.

rule:

$$\frac{\models [I]\ \textbf{if}\ (b)\ \{C\}\ [I] \qquad \models [I]\ \textbf{assume}\ \neg b\ [Q] \qquad \text{no } \forall\langle\_\rangle \text{ after any } \exists \text{ in } I}{\models [I]\ \textbf{while}\ (b)\ \{C\}\ [Q]}\ \text{WHILEUNSOUND-}\forall^*\exists^*$$

The key difference with the rule WHILE-∀*∃* is that the syntactic restriction is checked on the loop invariant I instead of the postcondition Q. Unfortunately, this alternative rule (and the encoding based on it) is unsound, as illustrated by the following example.

Example 6.4.2 The naive encoding based on the rule WHILEUNSOUND-∀*∃* is unsound.
Consider the method `naive_encoding` from Figure 6.10.Depending on the value of t, this method will either loop forever (if $t = 1$) or simply increment x until $x = n$ (if $n \geq 0$ and $t = 2$). Our precondition [15] requires the existence of a state that will loop forever ($t = 1$), and, for all possible non-negative values v of n, the existence of a state that will do n iterations until $x = n$. The postcondition, which does *not* hold, ensures the existence of an upper bound v for the value of x in all states.[16]

To understand why this postcondition does not hold, consider a set of states S that satisfies the precondition, *i.e.*, S contains at least one state with $t = 1$, and, for each natural number v, S contains a state σ where $\sigma(n) = v$ and $\sigma(t) = 2$. Moreover, let S' be the set of states after the loop. For all states with $t = 2$, x will be equal to n after the while loop, *i.e.*, $\forall \sigma' \in S'.\ \sigma'(t) = 2 \Rightarrow \sigma'(x) = \sigma'(n)$. Thus, for each natural number v, S' contains a state σ' where $\sigma'(x) = \sigma'(n) = v$ (and $\sigma'(t) = 2$). In particular, this implies that the set of values $\{\sigma'(x) \mid \sigma' \in S'\}$ does not have an upper bound. This contradicts the postcondition, which expresses the existence of such an upper bound.

We now explain why the naive encoding described above accepts this program. The first premise of the rule WHILEUNSOUND-∀*∃*, $\models [I]\ \textbf{if}\ (t = 1 \vee x < n)\ \{x := x + 1\}\ [I]$, holds, since any state σ_1 with $t = 1$ will always enter the if-branch, and thus σ_1 will keep having the maximal value (among all executions) for x. Moreover, the loop invariant I satisfies the syntactic restriction (no $\forall\langle\_\rangle$ appears under any existential quantifier), and I clearly holds before the loop, since $x = 0$ in all states.

15: Note that the second conjunct of the precondition is not required for the naive encoding to be unsound on this example, but we use it to convey more intuition.

16: In essence, this example is similar to Example 5.6.4: The two conjuncts of our invariant ensure the existence of an upper bound ($\sigma_1(x)$) for the value of x in all states.

Finally, let us consider what happens after the loop, and why this encoding allows us to derive the wrong postcondition. Let S be a set of states that satisfies the loop invariant I, and let S' be the set of states obtained by executing **assume** $\neg(t = 1 \vee x < n)$ in all states from S. Note that S' corresponds to the subset of states from S that satisfy $t \neq 1 \wedge x \geq n$. From I, we learn that there is a state σ_1 in S where $t = 1$, and that this state σ_1 has the maximum value for x among all states in S. Thus, there exists an upper bound v for the value of x in all states from S, namely $v \triangleq \sigma_1(x)$. Since S' is a subset of S, this upper bound v is also an upper bound for the value of x in all states from S', which corresponds to the invalid postcondition.

A new rule for automating $\forall^*\exists^*$-hyperproperties. The previous example shows that deriving the postcondition Q *semantically* from I, while checking the *syntactic* restriction on the loop invariant I, is unsound (*i.e.*, the rule WhileUnsound-$\forall^*\exists^*$ is unsound). Our novel loop rule WhileAuto-$\forall^*\exists^*$ solves this issue by deriving the postcondition Q *syntactically* from I while enforcing the *syntactic* restriction on I. This rule can be automated in a straightforward way, as shown by Figure 6.8c. We obtain the postcondition from I, which we write $\Theta_{\neg b}(I)$, by recursively replacing all instances of $\exists\langle\sigma\rangle.\, P$ with $\exists\sigma.\, P \wedge (\neg b(\sigma) \Rightarrow \langle\sigma\rangle)$.[17] That is, the postcondition ensures that the existentially-quantified states in I *exist*, but they are not guaranteed to belong to the set of states after the loop: they belong to the set of states after the loop if they satisfy the negation $\neg b$ of the loop guard. Although $\Theta_{\neg b}(I)$ is not a well-formed hyper-assertion according to the syntax in Section 6.2.3, this is not an issue since $\Theta_{\neg b}(I)$ is not an annotation in a user-written program but only appears in the generated Viper program.

17: We overload the notation $\langle\sigma\rangle$ to mean $\lambda S.\, \sigma \in S$, *i.e.*, the formula is equivalent to $\lambda S.\, \exists\sigma.\, P \wedge (\neg b(\sigma) \Rightarrow \sigma \in S)$.

Example 6.4.3 The rule WhileAuto-$\forall^*\exists^*$ correctly rejects the invalid example from Figure 6.10.
Using our new rule on the example from Figure 6.10, we obtain from I the postcondition $\Theta_{\neg b}(I) = (\exists\sigma_1.\, \sigma_1(t) = 1 \wedge (\neg(\sigma_1(t) = 1 \vee \sigma_1(x) < \sigma_1(n)) \Rightarrow \langle\sigma_1\rangle)) \wedge (\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(t) = 1 \Rightarrow \sigma_1(x) \geq \sigma_2(x))$, which does not entail the wrong postcondition anymore. Indeed, while we still learn the existence of a state σ_1 where $t = 1$, we do not learn that σ_1 belongs to the set of states after the loop, because we cannot prove $\neg(\sigma_1(t) = 1 \vee \sigma_1(x) < \sigma_1(n))$, and, thus, we cannot conclude that $\forall\langle\sigma_2\rangle.\, \sigma_1(x) \geq \sigma_2(x)$.

We have proven in Isabelle/HOL [33] that this novel rule, which Hypra leverages, is sound:

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

Theorem 6.4.1 Soundness of the novel loop rule for $\forall^*\exists^*$-hyperproperties. *The rule* WhileAuto-$\forall^*\exists^*$ *in Figure 6.7 is sound.*

Proof. To prove this result, we use the fact that $sem(\textbf{while}\ (b)\ \{C\}, S)$, the semantics of the while loop given a set of initial states S, can be seen as the limit of $sem([\textbf{if}\ (b)\ \{C\}]^n; \textbf{assume}\ \neg b, S)$ as n goes to infinity, where $[\textbf{if}\ (b)\ \{C\}^n]$ represents the statement $\textbf{if}\ (b)\ \{C\}$ sequentially composed

with itself n times. More formally:

$$sem(\textbf{while}\ (b)\ \{C\}, S) = \bigcup_{n\in\mathbb{N}} sem([\textbf{if}\ (b)\ \{C\}]^n; \textbf{assume}\ \neg b, S) \qquad (*)$$

In other words, every state after the loop must have exited the loop after n iterations, for some n. In particular, note that the sequence of sets $(sem([\textbf{if}\ (b)\ \{C\}]^n; \textbf{assume}\ \neg b, S))_{n\in\mathbb{N}}$ is non-decreasing. That is, the set of states that exit the loop in the first n iterations can only grow when n grows.

We then prove the following property $\mathcal{P}(I)$, by structural induction over the syntactic hyper-assertion I: "For any non-decreasing sequence $(S_n)_{n\in\mathbb{N}}$ of set of states, if I contains no $\forall\langle\_\rangle$ after any $\exists$, and if $\forall n.\, S_n \models \Theta_{\neg b}(I)$, then $(\bigcup_n S_n) \models \Theta_{\neg b}(I)$ holds". The theorem follows from this property and the aforementioned identity (*). In the following, we discuss four cases of the induction; all other cases are trivial.

Cases $\mathcal{P}(\exists y.\, I)$ and $\mathcal{P}(\exists\langle\sigma\rangle.\, I)$. These cases are straightforward: From the syntactic restriction, we know that I contains no $\forall\langle\_\rangle$, and so does $\Theta_{\neg b}(I)$. Intuitively, this means that $\Theta_{\neg b}(I)$ only cares about the *existence* of states, and thus $\Theta_{\neg b}(I)$ grows monotonically (which can be proven by an additional trivial induction on I): If it is satisfied by a set of states, then it will be satisfied by any superset of this set. Since it is satisfied by all S_n, it is also satisfied by their union.

Case $\mathcal{P}(\forall\langle\sigma\rangle.\, I)$. We assume (1) $\mathcal{P}(I)$ and (2) $\forall n.\, S_n \models \forall\langle\sigma\rangle.\, \Theta_{\neg b}(I)$, and we want to prove $(\bigcup_n S_n) \models \forall\langle\sigma\rangle.\, \Theta_{\neg b}(I)$. To prove this, let σ be a state in $\bigcup_n S_n$, and let us prove that $(\bigcup_n S_n), \sigma \models \Theta_{\neg b}(I)$ (which informally means that the previously existentially-quantified state σ is instantiated in $\Theta_{\neg b}(I)$ to the concrete state).[18] Because $\sigma \in \bigcup_n S_n$, there exists a k such that $\sigma \in S_k$. Let S' such that $\forall n.\, S'_n \triangleq S_{n+k}$. Because $\forall n.\, S'_n, \sigma \models \Theta_{\neg b}(I)$, we can use the induction hypothesis $\mathcal{P}(I)$ to get that $(\bigcup_n S'_n) \models \Theta_{\neg b}(I)$. Finally, notice that $(\bigcup_n S_n) = (\bigcup_n S'_n)$, because S is non-decreasing, which concludes the case.

18: In our mechanization, we use de Bruijn indices to handle quantifiers, which we ignore in this chapter for simplicity.

Case $\mathcal{P}(I_1 \vee I_2)$. We assume (1) $\mathcal{P}(I_1)$, (2) $\mathcal{P}(I_2)$, and (3) $\forall n.\, S_n \models \Theta_{\neg b}(I_1) \vee \Theta_{\neg b}(I_2)$, and we want to prove $(\bigcup_n S_n) \models \Theta_{\neg b}(I_1) \vee \Theta_{\neg b}(I_2)$. By (3), we know that $\Theta_{\neg b}(I_1) \vee \Theta_{\neg b}(I_2)$ is true infinitely often, which implies that either $\Theta_{\neg b}(I_1)$ is true infinitely often, or $\Theta_{\neg b}(I_2)$ is true infinitely often.[19] Without loss of generality, let us assume that $\Theta_{\neg b}(I_1)$ is true infinitely often, and let S' be an infinite subsequence of S such that $\forall n.\, S'_n \models \Theta_{\neg b}(I_1)$. By the induction hypothesis $\mathcal{P}(I_1)$, we get that $(\bigcup_n S'_n) \models \Theta_{\neg b}(I_1)$. Moreover, because S is non-decreasing, we get that $(\bigcup_n S_n) = (\bigcup_n S'_n)$, which concludes the case. □

19: Note that this argument only holds because a disjunction has a *finite* number of disjuncts (namely two). In contrast, this argument does not apply to existential quantifiers (which is why we need the syntactic restriction), as they informally have infinitely many disjuncts.

6.4.3. Automatic Framing

In the previous subsections, we have shown how we derived verification conditions from the loop rules offered by Hyper Hoare Logic. However, using those loop rules on their own (and not in conjunction with other rules as we show below) has the limitation that only the information

```
method framing1(x: Int) returns (y: Int)
    requires ∀⟨σ1⟩, ⟨σ2⟩. σ1(x) = σ2(x)
    requires ∀⟨σ⟩. σ(x) ≥ 0
    ensures ∀⟨σ1⟩, ⟨σ2⟩. σ1(y) = σ2(y)
{
    y := 0
    while (y < x)
        invariant ∀⟨σ⟩. σ(y) ≤ σ(x)

    { y := y + 1 }
}
```

```
method framing2(x: Int) returns (y: Int)
    requires ∃⟨σ⟩. ∀⟨σ'⟩. σ(x) ≥ σ'(x)
    requires ∀⟨σ⟩. σ(x) ≥ 0
    ensures ∃⟨σ⟩. ∀⟨σ'⟩. σ(y) ≥ σ'(y)
{
    y := 0
    while (y < x)
        invariant ∀⟨σ⟩. σ(y) ≤ σ(x)
        decreases x - y
    { y := y + 1 }
}
```

Figure 6.11.: An example from Hypra that requires automatic framing to be successfully verified.

provided by the loop invariant is preserved, as we illustrate with the examples in Figure 6.11.

Example 6.4.4 The intuitive loop invariant is not enough.
Consider the method `framing1` on the left of the figure, which increments y in a loop until $x = y$. We want to prove that if x has the same initial value in all executions, then y will have the same final value in all executions. Using the standard (unary) loop invariant $\forall\langle\sigma\rangle.\, \sigma(y) \leq \sigma(x)$, we can easily prove that, after the loop, $x = y$ in all states (1). Moreover, since all executions have the same value for x before the loop, and since x is not modified by the loop, all executions will still have the same value for x after the loop (2). By conjoining (1) and (2), we get the postcondition.

Unfortunately, the loop encodings presented so far are only able to prove (1), but not (2), since they assume only (a property derived from) the loop invariant after the loop. Because our loop invariant does not mention that x has the same value in all executions, this piece of information is lost after the loop.

One way to solve this particular problem is to add this information to the loop invariant, by conjoining $\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(x) = \sigma_2(x)$ to it. This solution is cumbersome for the user, who is required to write longer invariants, by adding information not relevant for the loop (but only for the postcondition later).

Another way to solve this issue is to use the following compositionality rule from Hyper Hoare Logic (where $mod(C)$ represents the variables modified by C and $fv(F)$ the (program) variables that appear in F), which allows propagating information about variables not modified by the loop after the loop:

$$\textsc{FrameSafe}\ \frac{\models [P]\ C\ [Q] \qquad \text{no } \exists\langle\_\rangle \text{ in } F \qquad mod(C) \cap fv(F) = \varnothing}{\models [P \wedge F]\ C\ [Q \wedge F]}$$

Since, in Example 6.4.4, x is not modified by the loop, we can use this rule with $F \triangleq (\forall\langle\sigma_1\rangle, \langle\sigma_2\rangle.\, \sigma_1(x) = \sigma_2(x))$ to solve our issue. Our goal is to *automatically* use this rule to frame information around the loop, without requiring the user to provide F.

We achieve this by adding (after the loop) the assumption that, for each

state in the set of states after the loop, there must exist a state (before the loop) with the same values for all variables not modified by C:

```
... // loop encoding
assume ∀σ_j ∈ S^j_∀. ∃σ_i ∈ S^i_∀. (∀x ∉ fv(C). σ_i(x) = σ_j(x)) // (F-OX)
```

where $S^i_\forall$ is the lower bound of the set of states *before* the loop, and $S^j_\forall$ is the lower bound of the set of states *after* the loop.

Intuitively, adding this assumption is sound because every state σ_j after the loop corresponds to the final state of an execution of C in an initial state σ_i from before the loop, and thus σ_i and σ_j must have the same values for the variables not modified by C. Formally, this encoding is justified by the following straightforward lemma:

Lemma 6.4.2 Soundness of the framing encoding.
$\forall \sigma' \in \mathit{sem}(C, S). \exists \sigma \in S. \forall x \notin \mathit{mod}(C). \sigma(x) = \sigma'(x)$

This encoding is stronger than *any* possible application of the rule FrameSafe, since the former logically implies the latter (for any frame F). To see why it solves the issue from Example 6.4.4, consider two states σ'_1 and σ'_2 that belong to the set of states $S^j_\forall$ after the loop. From the assumption (F-OX), we get the existence of two states σ_1 and σ_2 from $S^i_\forall$, such that $\sigma_1(x) = \sigma'_1(x)$ and $\sigma_2(x) = \sigma'_2(x)$. Because of the precondition, we know that $\sigma_1(x) = \sigma_2(x)$, and thus can conclude that $\sigma'_1(x) = \sigma'_2(x)$.

Framing hyperproperties with existentially-quantified states. Note that the rule FrameSafe has the restriction that F is not allowed to existentially quantify over states. This is a limitation, as shown by the following example.

Example 6.4.5 A limitation of the rule FrameSafe.
Consider the method `framing2` on the right of Figure 6.11. This method has the same body and loop invariant as method `framing1`, but we now want to prove that if there is an execution whose initial value x is maximal (among all executions), then there should exist an execution whose final value for y is also maximal. In this case, we would like to apply the rule FrameSafe with the frame $F \triangleq (\exists \langle \sigma \rangle. \forall \langle \sigma' \rangle. \sigma(x) \geq \sigma'(x))$, but cannot because this frame F contains an existential quantifier over states.

To overcome this limitation, Hyper Hoare Logic provides the following rule, which lifts this restriction:

$$\textsc{Frame}\quad \frac{\models_{\Downarrow} [P]\ C\ [Q] \qquad \begin{array}{c} \mathit{mod}(C) \cap \mathit{fv}(F) = \varnothing \\ F \text{ is a syntactic hyper-assertion} \end{array}}{\models_{\Downarrow} [P \wedge F]\ C\ [Q \wedge F]}$$

This rule requires however to prove a stronger triple, the *terminating* hyper-triple $\models_{\Downarrow} [P]\ C\ [Q]$ (formally defined in Section 5.3.2), which must ensure the existence of a terminating execution from any initial state. In Hypra, we can ensure that a triple around a loop **while** (b) $\{C\}$ is terminating as long as C contains no **assume** statements, and this loop

and all nested loops in C terminate, *i.e.*, have been annotated with a **decreases** clause. This is for example the case for the loop in method framing2. When those two conditions hold, it is sound to strengthen the previous encoding with the additional assumption (F-UX), as follows:

```
S_p := S
... // loop encoding
assume ∀σ_j ∈ S^j_∀. ∃σ_i ∈ S^i_∀. (∀x ∉ fv(C). σ_i(x) = σ_j(x)) // (F-OX)
assume ∀σ_i ∈ S^i_∃. ∃σ_j ∈ S^j_∃. (∀x ∉ fv(C). σ_i(x) = σ_j(x)) // (F-UX)
```

Together, those two assumptions are stronger than the application of the rule Frame for any frame F. For example, emitting those two assumptions together lets us automatically derive that $\exists\langle\sigma\rangle.\, \forall\langle\sigma'\rangle.\, \sigma(x) \geq \sigma'(x)$ holds after the loop in our example. However, we have noticed in practice that emitting the assumption (F-UX) does not interact well with the encoding described in Section 6.3.2, and might result in matching loops. Thus, Hypra provides an option to emit this second assumption (when applicable), which is disabled by default.

Framing hyperproperties inside loop iterations. Finally, note that it is sound to emit the assumption (F-OX) at the beginning of an arbitrary loop iteration, which can lead to more concise invariants. Moreover, it is sound to emit assumption (F-UX) as well, if the loop body C contains no **assume** statements, and loops nested within C terminate (but the outer loop is not required to terminate). In practice, our verifier can emit those assumptions, but this also requires *inlining* the verification of the loop body with the loop invariant inside the method containing the outer loop (as opposed to verifying the loop body with the loop invariant in a separate Viper method), which can worsen performance. Thus, our verifier does not do it by default, but provides an option to do it.

6.5. Implementation and Evaluation

We implemented Hypra, a deductive program verifier for hyperproperties, on top of Viper; that is, Hypra takes as input a text file, translates it into a Viper program (as described in Section 6.3), calls the Viper verifier to verify this program, and then translates the output (successful verification or error messages) back to the user.[20]

We evaluated Hypra on a diverse set of examples, which includes many examples from the literature, to answer the following questions:

(RQ1) Can Hypra (dis-)prove hyperproperties of different types, namely $\forall^*$, $\exists^*$, $\forall^*\exists^*$, and $\exists^*\forall^*$?

(RQ2) How many lines of proof annotations are needed by Hypra?

(RQ3) Can Hypra verify complex examples in a reasonable amount of time?

In summary, our evaluation shows that Hypra can efficiently (dis-)prove hyperproperties of different types with a reasonable amount of proof annotations, and it can do so within a reasonable amount of time. In the following, we describe how we selected our benchmarks, how we ran the experiments, and present and discuss the results.

20: As explained in Chapter 2, Viper actually provides two back-end verifiers, one based on symbolic execution, and one based on Boogie [12]. Hypra uses the two Viper verifiers to verify the generated Viper program, and reports the first successful verification result.

Benchmarks. To evaluate Hypra, we used the benchmarks from HyPa [128], Descartes [88], ORHLE [96], and PCSat [127]: We selected a subset of their publicly-available benchmarks and translated them into the programming language supported by Hypra to form our benchmarks. We selected the benchmarks based on the following criteria:

1. For Descartes, ORHLE, HyPa, and PCSat, we ignored the benchmarks that use data structures not supported by Hypra, such as arrays.[21]
2. For Descartes, we ignored the benchmarks that use objects with more than 3 fields, since translating fields into the language supported by Hypra is cumbersome.
3. For ORHLE, HyPa, and PCSat, we ignored the benchmarks that prove *relational* properties (*i.e.*, properties relating multiple executions of *different* programs), since Hypra only supports *hyperproperties*.
4. For PCSat, we selected only the benchmarks that do not require reasoning about co-termination, since Hypra does not support this.

For each selected benchmark, we translated it to the syntax accepted by Hypra. To obtain hyper-triples semantically equivalent to the original specifications, we used the formal translations presented in Appendix A.3. In addition, we annotated the translated benchmarks with loop variants, loop invariants and hints when necessary.

We additionally created new benchmarks, by taking existing benchmarks that fail to prove $\forall^*$- or $\forall^*\exists^*$-hyperproperties, and formally proving that they violate these hyperproperties. To do so, we strengthened the preconditions and proved the negation of the original postconditions, following Theorem 5.3.4. In particular, this allows us to obtain benchmarks with $\exists^*\forall^*$-hyperproperties, which are not included in the benchmark suites we draw from.

In total, we obtained 84 benchmarks. Figure 6.1 provides more details about the selected benchmarks.

Experimental Setup. We ran Hypra to verify the translated benchmarks on a MacBook Pro running macOS Ventura 13.3 with a 2.3 GHz 8-Core Intel Core i9 processor and 32 GB RAM. Each benchmark was run with 10 repetitions. For each run, we recorded the verification result and runtime. In the end, we checked that the verification results in all runs were consistent, and also computed the average verification time for each benchmark.

Results. The results of our evaluation are shown in Figure 6.1. As we can see, Hypra can handle not only all $\forall^*-$, $\exists^*$- and $\forall^*\exists^*$-hyperproperties that other verifiers can handle, but also $\exists^*\forall^*$-hyperproperties, which no other existing verifier supports.

Although verification using Hypra is not fully automatic, it only requires a reasonable amount of proof annotations from users, which is evidenced by the last column of Figure 6.1.

Moreover, Hypra is quite efficient in general. On average, it took Hypra 258 seconds to run the entire benchmark suite composed of 84 programs.

[128]: Beutner et al. (2022), *Software Verification of Hyperproperties Beyond K-Safety*

[88]: Sousa et al. (2016), *Cartesian Hoare Logic for Verifying K-Safety Properties*

[96]: Dickerson et al. (2022), *RHLE*

[127]: Unno et al. (2021), *Constraint-Based Relational Verification*

21: We have since extended Hypra to support mathematical sequences [239].

Table 6.1.: Results of our evaluation. Benchmarks marked with † are obtained by strengthening the preconditions and negating the postconditions of the original benchmarks that fail to prove ∀* or ∀*∃*-hyperproperties. We count **use** statements, loop variants and loop invariants as annotations.

Type of hyperproperty	Source	Files		Verification time		Annotations
		no.	Mean (LoC)	Mean (s)	Median (s)	Mean (LoC)
∀*	Descartes	15	129	2.3	1.7	0.0
	PCSat	3	23	1.1	1.1	2.7
	Overall	**18**	**111**	**2.1**	**1.6**	**0.4**
∃*	Descartes†	8	81	13.0	5.0	0.0
	ORHLE	6	29	2.9	2.5	7.7
	Overall	**14**	**59**	**8.7**	**3.5**	**3.3**
∀*∃*	ORHLE	28	20	2.3	1.4	1.2
	HyPa	8	14	1.2	1.1	2.1
	PCSat	1	22	1.2	1.2	2.0
	Overall	**37**	**19**	**2.0**	**1.3**	**1.4**
∃*∀*	ORHLE†	15	25	1.6	1.2	1.7

For 93% of the benchmarks, verification finished within 5 seconds. In some rare cases, the runtime was relatively long, with the maximum runtime around 35 seconds. This is not unexpected, since some of those benchmarks have very complex commands (such as lots of nested conditional statements) and specifications (such as preconditions and postconditions of the shape ∃∃∃∀∀∀).

In summary, our evaluation demonstrates that Hypra can effectively verify hyperproperties of different types with a reasonable amount of proof annotations and within a reasonable amount of time.

6.6. Related Work

In this section, we only cover tools and approaches for automatically verifying hyperproperties, as program logics for hyperproperties have already been discussed in Section 5.8.

Deductive Verification. As explained in Chapter 1, deductive verifiers are tools that, given as input a program, a specification, and proof hints (such as loop invariants), try to automatically construct a proof in a given program logic that the program satisfies the specification. Many deductive verifiers based on SMT solvers (such as Z3 [52]) have been developed for verifying *safety properties*, *i.e.*, properties that should hold for all *individual* executions, such as Boogie [12], Why3 [17], Dafny [13], or Viper [16].

The problem of verifying that a program satisfies a k-safety hyperproperty can be reduced to the problem of verifying that a product program [122, 123, 216] satisfies a safety property, where the product program is for example obtained by composing sequentially k renamed copies of the original program. The product program can then be verified using deductive verifiers tailored for safety properties. Eilers et al. [124] show how to treat method calls modularly in this context, allowing methods to have relational preconditions and postconditions, similar to the ∀*-specifications

[52]: de Moura et al. (2008), *Z3*

[12]: Leino (2008), *This Is Boogie 2*

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

[13]: Leino (2010), *Dafny*

[16]: Müller et al. (2016), *Viper*

[122]: Barthe et al. (2004), *Secure Information Flow by Self-Composition*

[123]: Barthe et al. (2011), *Relational Verification Using Product Programs*

[216]: Terauchi et al. (2005), *Secure Information Flow as a Safety Problem*

[124]: Eilers et al. (2019), *Modular Product Programs*

shown in Section 6.2 (for example in Figure 6.1). Barthe et al. [125] present an asymmetric construction for product programs, which allows proving relational ∀∃-properties such as program refinement [240].

Deductive verifiers specifically targeting hyperproperties have been developed as well. Those include WhyRel [126], SecC (based on SecCSL) [114], and HyperViper (based on CommCSL) [115] for non-interference [208] (a 2-safety hyperproperty), Descartes (based on Cartesian Hoare Logic) [88] for k-safety hyperproperties, and ORHLE (based on RHLE) [96] for $\forall^*\exists^*$-hyperproperties. As our evaluation shows, our tool handles well the benchmarks from Descartes and ORHLE, and can even disprove invalid ones. Another recent verifier for $\forall^*\exists^*$-hyperproperties is ForEx [121]. Compared to ORHLE and ForEx, the closest to our work, our tool Hypra is more expressive, since it also supports for example $\exists^*\forall^*$-hyperproperties, and supports reasoning about runtime errors. Our tool is also more flexible, since it allows the user to write explicit quantifiers in the assertion language itself, and thus allows one to combine different types of hyperproperties in the same proof, whereas ORHLE and ForEx require the user to fix the quantification scheme in advance. Moreover, even for $\forall^*\exists^*$-hyperproperties, our tool supports reasoning about more complex proof patterns, such as while loops where different executions might exit at different iterations.

Other approaches have been developed to automatically verify hyperproperties [127, 204, 233, 241, 242]. For example, Assaf et al. [204] use abstract interpretation [234] to verify different hypersafety properties related to information flow, including some safety hyperproperties that are not k-safety for any k. To achieve this, they present a hypercollecting semantics, similar in spirit to the function *sem* (Definition 6.2.2) from Hyper Hoare Logic. Unno et al. [127] present PCSat, a tool based on a generalization of *constrained Horn clauses* [243] to automatically verify k-safety hyperproperties, and more complex hyperproperties such as termination-sensitive non-interference [244] and generalized non-interference [210, 211]. As shown in our evaluation, our tool Hypra can handle all the benchmarks from PCSat that fall in our supported subset of programs, with a reasonable amount of proof annotations and in reasonable time. Extending Hypra to reason about properties such as termination-sensitive non-interference is future work.

Finally, *temporal logics* to express hyperproperties have been proposed, such as *HyperLTL* and *HyperCTL** [229], and *model checking* [245] algorithms to check whether finite-state systems satisfy hyperproperties expressed in these temporal logics have been proposed [246]. For example, Hsu et al. [231] have proposed algorithms for *bounded* model checking, Coenen et al. [230] proposed model checking algorithms for $\forall^*\exists^*$-hyperproperties, and Beutner and Finkbeiner [232] proposed an explicit-state model checking algorithm that is complete for HyperLTL and for hyperproperties with arbitrary quantifier alternations. Beutner and Finkbeiner [128] have also shown that model checking techniques for $\forall^*\exists^*$ can be applied to infinite-state systems, by using predicate abstraction.

[125]: Barthe et al. (2013), *Beyond 2-Safety*

[240]: Abadi et al. (1991), *The Existence of Refinement Mappings*

[126]: Nagasamudram et al. (2023), *The WhyRel Prototype for Modular Relational Verification of Pointer Programs*

[114]: Ernst et al. (2019), *SecCSL*

[115]: Eilers et al. (2023), *CommCSL*

[208]: Volpano et al. (1996), *A Sound Type System for Secure Flow Analysis*

[88]: Sousa et al. (2016), *Cartesian Hoare Logic for Verifying K-Safety Properties*

[96]: Dickerson et al. (2022), *RHLE*

[121]: Beutner (2024), *Automated Software Verification of Hyperliveness*

[127]: Unno et al. (2021), *Constraint-Based Relational Verification*

[204]: Assaf et al. (2017), *Hypercollecting Semantics and Its Application to Static Analysis of Information Flow*

[233]: Barthe et al. (2019), *Verifying Relational Properties Using Trace Logic*

[241]: Farzan et al. (2019), *Automated Hypersafety Verification*

[242]: Itzhaky et al. (2024), *Hyperproperty Verification as CHC Satisfiability*

[204]: Assaf et al. (2017), *Hypercollecting Semantics and Its Application to Static Analysis of Information Flow*

[234]: Cousot et al. (1977), *Abstract Interpretation*

[127]: Unno et al. (2021), *Constraint-Based Relational Verification*

[243]: Bjørner et al. (2015), *Horn Clause Solvers for Program Verification*

[244]: Volpano et al. (1997), *Eliminating Covert Flows with Minimum Typings*

[210]: McCullough (1987), *Specifications for Multi-Level Security and a Hook-Up*

[211]: McLean (1996), *A General Theory of Composition for a Class of "Possibilistic" Properties*

[229]: Clarkson et al. (2014), *Temporal Logics for Hyperproperties*

[245]: Clarke (1997), *Model Checking*

[246]: Finkbeiner et al. (2015), *Algorithms for Model Checking HyperLTL and HyperCTL $*$*

[231]: Hsu et al. (2021), *Bounded Model Checking for Hyperproperties*

[230]: Coenen et al. (2019), *Verifying Hyperliveness*

[232]: Beutner et al. (2023), *AutoHyper*

[128]: Beutner et al. (2022), *Software Verification of Hyperproperties Beyond K-Safety*

7. Conclusion

S'agirait de grandir hein, s'agirait de grandir...

Hubert Bonisseur de la Bath, *OSS 117 : Le Caire, nid d'espions*

In this thesis, we have addressed two key challenges faced by modern verifiers: trustworthiness and expressiveness. In the first part, we have addressed the trustworthiness challenge by developing novel formal foundations to justify the soundness of automated verifiers based on separation logic, in particular for translational verifiers (Chapter 2), fractional predicates (Chapter 3), and magic wands (Chapter 4). In the second part, we have addressed the expressiveness challenge by developing a novel program logic for hyperproperties, Hyper Hoare Logic (Chapter 5), which can express hyperproperties that no other program logic supports, and by developing a novel automated verifier for hyperproperties, Hypra (Chapter 6), based on Hyper Hoare Logic.

Our work highlights the importance of conducting research at the intersection of theory and automation for two key reasons, which we discuss in Section 7.1: (1) automated verifiers require dedicated *practical theory* (as opposed to what we call *pure theory*, *i.e.*, theory developed without automation in mind), and (2) insights from automation and practical theory lead to new formal results that enrich pure theory. We then discuss promising future work directions in Section 7.2.

7.1. Research at the Intersection of Theory and Automation

Automated verifiers need dedicated practical theory

While *pure theory* provides the basis for automation (*e.g.*, separation logic provides the basis for Viper, and Hyper Hoare Logic provides the basis for Hypra), it does not address many challenges faced by automated verifiers, which require dedicated *practical theory*. For example, intermediate verification languages (Chapter 2) are primarily motivated by practical purposes such as reusing optimized automation across different verifiers, and CoreIVL's operational semantics is motivated by the connection to the back-end verifiers. Similarly, automated verifiers support fractional predicates (Chapter 3) with the *syntactic* multiplication rather than the *semantic* one, because the former is straightforward to automate.

Moreover, one of the key goals in the design of automated verifiers is to minimize the number of hints required from the user, a requirement that is not considered by pure theory. For example, package algorithms for magic wands (Chapter 4) only make sense in the context of automation. In pure theory, one can simply manually provide the footprint and the

corresponding proof. Similarly, Hypra's (Chapter 6) new rule WhileAuto-$\forall^*\exists^*$ for loops (which requires *one* user-provided hyper-assertion only) is advantageous compared to HHL's rule While-$\forall^*\exists^*$ (which requires *two* user-provided hyper-assertions) in the context of automated verifiers only, as it reduces the number of required user-provided hints. The motivation for the framing encoding presented in Section 6.4.3 is likewise driven by the goal of reducing hints, as it reduces the size and complexity of the required user-provided loop invariant. In contrast, in pure theory, the same effect can be achieved with a single application of the FrameSafe rule, which does not require such encoding.

Theory benefits from insights from automation

Building automated verifiers based on pure theory naturally leads to new insights that benefit pure theory, as applying verifiers to large programs allows us to identify gaps and limitations of existing theory. As anecdotal evidence, the important limitation of HHL's core rule Iter for loops only became clear to us as we were trying out some examples during the early development of Hypra (specifically, an example similar to Example 5.6.3).

Perhaps surprisingly, practical theory (which primarily addresses challenges specific to automation) also yields insights that advance pure theory. For instance, the verification primitives `inhale` and `exhale` (Chapter 2), introduced by Leino and Müller [152] for automation, have proven valuable in pure theory, to prove refinement [171]. Similarly, the angelism in CoreIVL's operational semantics, originally used to abstract over verification algorithms and first applied in an SL context to describe VeriFast's symbolic execution [67], has proven useful in pure theory, to model interactions between programs written in different languages [154, 170]. Implicit dynamic frames (IDF) [70], for which we have provided a novel foundation via an IDF algebra (Chapter 2), were originally motivated by practical concerns, namely letting programmers write specifications in the programming language's syntax, and facilitating automation via verification condition generation. Recently, IDF have been integrated [172] into Iris [31, 41], to reduce redundancy between specifications and implementations. As yet another example, the syntactic multiplication for fractional predicates (Chapter 3), primarily motivated by its amenability to automation, yields a formal semantics with better theoretical properties (such as distributivity, factorizability, and combinability) compared to existing pure theory [83, 84].

[152]: Leino et al. (2009), *A Basis for Verifying Multi-threaded Programs*

[171]: Song et al. (2023), *Conditional Contextual Refinement*

[67]: Jacobs et al. (2015), *Featherweight VeriFast*

[154]: Guéneau et al. (2023), *Melocoton*

[170]: Sammler et al. (2023), *DimSum*

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

[172]: Spies et al. (2025), *Destabilizing Iris*

[31]: Jung et al. (2018), *Iris from the Ground Up*

[41]: Jung et al. (2015), *Iris*

[83]: Le et al. (2018), *Logical Reasoning for Disjoint Permissions*

[84]: Brotherston et al. (2020), *Reasoning over Permissions Regions in Concurrent Separation Logic*

The role of syntax

Another important insight from this thesis is the importance of syntax. Automated verifiers *require* a formal syntax for assertions, as users write specifications in concrete syntax, which are then parsed by verifiers. But beyond this necessity, we have found that a carefully designed assertion language brings several theoretical and practical benefits.

Universal properties for free. There are infinitely fewer syntactic assertions than semantic ones, as there is a countable number of syntactic assertions, while there are uncountably many semantic assertions [247]. While this may seem like a limitation (as infinity is often considered *very* large), syntactic assertions provide a major benefit: they automatically satisfy desirable properties that are difficult (or even impossible) to guarantee for arbitrary semantic assertions. For instance, because Viper's assertion language disallows negations and disjunctions of impure assertions (due to automation challenges), every assertion expressible in Viper is combinable (Chapter 3).[1] Similarly, all recursive predicate definitions that can be written in Viper are syntactically positive, and thus admit well-defined least and greatest fixed points.[2] As another example, HHL's rule Frame is sound for all syntactic hyper-assertions F, whereas it would be unsound for arbitrary semantic ones. Moreover, by carefully restricting syntax, we obtain semantic guarantees such as downward-closure (needed to justify the rule FrameSafe), or the soundness of the rules While-$\forall^*\exists^*$ and WhileAuto-$\forall^*\exists^*$, which rely on a non-trivial semantic property.

[247]: Cantor (1890/91), *Über eine elementare Frage der Mannigfaltigkeitslehre.*

1: Assuming that magic wands are interpreted in the unbounded logic from Chapter 3.

2: This is because Viper disallows magic wands inside predicate definitions.

Syntax facilitates automation. Because syntactic structures are explicitly available, automation can operate directly on the structure of assertions. For example, the package logic (Chapter 4) leverages the syntactic structure of assertions to deconstruct a magic wand's right-hand side into atomic parts, making it easier to infer the required resources for the footprint. Similarly, the automation of fractional predicates (Chapter 3) is straightforward, as it uses the syntactic multiplication, which operates on the syntax of assertions rather than their semantics. Syntax also enables powerful rules based on simple syntactic transformations (such as the HHL rules AssumeS, AssignS, HavocS, and Specialize from Chapter 5), which are easier to apply than their semantic counterparts.

7.2. Future Work

While this thesis advances the state of the art in both the trustworthiness and expressiveness of automated verifiers, it also opens up many interesting research directions for future exploration. Below, we outline future work along two main axes: enhancing the trustworthiness of SL-based verifiers (Section 7.2.1), and advancing automated hyperproperty verification (Section 7.2.2). Additional promising directions, which we do not cover here, include developing formal foundations for automated verifiers based on other program logics, such as Dafny [13] (which relies on dynamic frames [248]), and designing automated verifiers for probabilistic programs [249] and their corresponding probabilistic properties.

[13]: Leino (2010), *Dafny*

[248]: Kassios (2006), *Dynamic Frames*

[249]: Gordon et al. (2014), *Probabilistic Programming*

7.2.1. Trustworthiness of SL-based Verifiers

Full certification of existing SL-based verifiers

Our long-term goal, building on this thesis and the work of Parthasarathy [250], is to fully certify the soundness of practical automated verifiers

[250]: Parthasarathy (2024), *Formally Validating Translational Program Verifiers*

[16]: Müller et al. (2016), *Viper*

based on separation logic, such as Viper [16]. This requires several advances, which we outline below.

Extending ViperCore with advanced SL features. The first required step is to extend our ViperCore instantiation of CoreIVL (and the corresponding formalization and certification of VCGSem [74]) to support advanced SL features, including recursive predicates (Chapter 3), magic wands (Chapter 4), heap-dependent recursive functions [251], iterated separation conjunctions [151], higher-order predicates, and more. Supporting higher-order predicates, that is, predicates taking other predicates as arguments, may require incorporating step-indexing [191] into the CoreIVL model.

[74]: Parthasarathy et al. (2024), *Towards Trustworthy Automated Program Verifiers*

[251]: Heule et al. (2013), *Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions*

[151]: Müller et al. (2016), *Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution*

[191]: Appel et al. (2001), *An Indexed Model of Recursive Types for Foundational Proof-Carrying Code*

Adapting CoreIVL to support permission introspection. Another important step is adapting CoreIVL to support permission introspection features, provided in different forms by verifiers like Viper and VeriFast [15]. Viper's permission introspection expression `perm(x.f)`, where `x.f` is a heap location, yields the permission amount of `x.f` currently held by the Viper state. This allows programs to inspect and act on the amount of permission held, enabling advanced verification strategies and checks for permission leaks. However, as shown in work not presented in this thesis [73], this feature is challenging to formalize. In particular, it is not compatible with how CoreIVL models assertions, as it (1) behaves differently for `inhale` and `exhale`,[3] (2) makes the separating conjunction non-commutative,[4] and (3) has a non-local meaning (since the value of `perm(x.f)` depends on the program location).

[15]: Jacobs et al. (2011), *VeriFast*

[73]: Dardinier et al. (2023), *Verification-Preserving Inlining in Automatic Separation Logic Verifiers*

3: For example, the statement `exhale` $\mathbf{acc}(x.f) * \mathbf{perm}(x.f) > 0$ *never* succeeds, as $\mathbf{perm}(x.f)$ is evaluated after the permission to `x.f` is exhaled, and so it can only evaluate to 0. In contrast, the statement `inhale` $\mathbf{acc}(x.f) * \mathbf{perm}(x.f) > 0$ *always* succeeds, as $\mathbf{perm}(x.f)$ is evaluated after the permission to `x.f` is inhaled, so it must evaluate to 1.

4: For example, the assertions $\mathbf{acc}(x.f) * \mathbf{perm}(x.f) > 0$ and $\mathbf{perm}(x.f) > 0 * \mathbf{acc}(x.f)$ are not equivalent, as exhaling the latter could succeed (*e.g.*, in a state with full permission to `x.f`), while exhaling the former will never succeed, as explained above.

Handling the discrepancy of state models between front-ends and IVLs. To support realistic Viper front-ends, such as VerCors [57], Nagini [58], Prusti [59, 252], or Gobra [24], our approach must also be extended. For example, the approach in Chapter 2 connects a front-end with the same state model as our ViperCore instantiation, but in practice, front-end state models often differ substantially from the IVL state model.

[57]: Blom et al. (2017), *The VerCors Tool Set*

[58]: Eilers et al. (2018), *Nagini*

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

[252]: Astrauskas et al. (2022), *The Prusti Project*

[24]: Wolf et al. (2021), *Gobra*

A formal framework for the sound composition of transformations. Finally, many IVL features are implemented as *IVL-to-IVL* transformations, such as termination checking, method call inlining and loop unrolling [73], modular product program transformations [124], invariant inference [253], and more. The sound composition of these transformations is not always straightforward. It has for instance been shown that applying a modular product program transformation after a front-end translation is not always sound [124]. Developing a general formal framework for the sound composition of verification transformations remains an important open problem.

[73]: Dardinier et al. (2023), *Verification-Preserving Inlining in Automatic Separation Logic Verifiers*

[124]: Eilers et al. (2019), *Modular Product Programs*

[253]: Dohrau (2022), *Automatic Inference of Permission Specifications*

[124]: Eilers et al. (2019), *Modular Product Programs*

Extending SL-based verifiers

Beyond justifying the soundness of existing verifiers, formal foundations also pave the way for extending SL-based verifiers with new features, as discussed in Chapter 3 (*e.g.*, allowing magic wands inside fractional

predicates) and Chapter 4 (*e.g.*, adapting package algorithms to support combinable wands). Below, we describe several features that could be formalized and then added to existing SL-based verifiers.

Loop contracts and k-induction. While the standard way to verify loops in SL-based automated verifiers is to use inductive invariants, alternatives exist, such as k-induction [254] or loop contracts [174]. As we have proven sound the standard encoding of loops with inductive invariants based `inhale` and `exhale` in Chapter 2, we could formalize and prove sound, using CoreIVL, encodings for these alternative approaches, before integrating them into existing SL-based verifiers.

[254]: Donaldson et al. (2011), *Software Verification Using K-Induction*

[174]: Tuerk (2010), *Local Reasoning about While-Loops*

Magic wands with bipartite graphs. As discussed in Chapter 4, magic wands are useful for reasoning about Rust borrows (for example, in Prusti [59]), but are limited to one left-hand side and one right-hand side. To support more borrow patterns in Rust, one could generalize wands to bipartite graphs and adapt the package logic accordingly.

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

Universal introduction. A universal introduction feature could also be added to automated verifiers, simplifying the proof of universally quantified postconditions. For example, to prove a postcondition $\forall x.\, Q(x)$, one can fix a variable x with an arbitrary value at the start of the method, and prove $Q(x)$ at the end (instead of $\forall x.\, Q(x)$). This allows the use of x during the proof, for example to branch on its value.[5] One interesting use case for which this feature would be particularly helpful is the correctness of the Sheartsort algorithm for sorting matrices [255], which can be established with the 0-1 principle [256].[6] Another benefit is avoiding unnecessary quantifier instantiations, which would improve performance. However, such a feature must be carefully designed to be sound. For example, the value of x should not influence parts of the program that Q depends on, as well as termination.[7]

5: Note that this would go beyond the existing support for universal introduction in verifiers like F* [14] or Dafny [13], as they do not allow any side effect while performing the universal introduction.

[255]: Scherson et al. (1989), *Parallel Sorting in Two-Dimensional VLSI Models of Computation*

[256]: Knuth (1998), *The Art of Computer Programming, Volume 3*

6: The idea is to first fix an arbitrary value k, and then consider all values smaller than k to be 0, and all values larger than k to be 1.

7: While this sounds similar to the treatment of ghost code [257], the distinction between program code and ghost code does not really exist in IVLs. Even with this distinction, the assertion Q could express some property of a ghost computation, in which case the same issue arises.

7.2.2. Hyperproperty Verification

We believe that Hyper Hoare Logic (Chapter 5) offers a promising foundation for advancing automated hyperproperty verification, and opens several interesting and promising research directions, both in theory and automation, as outlined below.

Theory

Extending HHL to support relational properties for different programs. A main limitation of HHL, compared to other relational program logics, is that HHL only supports one program. That is, HHL only supports proving hyperproperties [27] (relating multiple executions of a program), as opposed to relational properties (relating executions of different programs). This is also a limitation when verifying hyperproperties, for example, when verifying a conditional statement where different executions take different branches. We believe that the ideas behind HHL could be extended to a relational setting.

[27]: Clarkson et al. (2008), *Hyperproperties*

Termination reasoning in HHL. Currently, HHL does not support proving termination,[8], nor non-termination. Similarly to how quantifiers over normal states in HHL can be used to reason about the presence and absence of executions, and how quantifiers over error states in Hypra can be used to reason about the presence and absence of errors, we believe that an extension of HHL to reason about termination and non-termination with quantifiers over non-terminating executions is possible, and would be useful for reasoning about other advanced properties such as termination-sensitive non-interference. Technically, such an extension would require using a different type of semantics than the big-step semantics we currently use, such as a small-step semantics, or the one used by Li et al. [258].

8: As discussed in Section 5.3.2, even terminating hyper-triples do not guarantee that all executions terminate.

[258]: Li et al. (2025), *Total Outcome Logic*

(Concurrent) Hyper Separation Logic. In its current version, HHL only supports simple programs without pointers or concurrency. A natural next step would be to extend HHL to *Hyper Separation Logic*, combining ideas from HHL with ideas from separation logic [10], to support local reasoning for programs that manipulate the heap. This extension could then be further developed into a concurrent version by adapting ideas from concurrent separation logic [11].

[10]: Reynolds (2002), *Separation Logic*

[11]: O'Hearn (2007), *Resources, Concurrency, and Local Reasoning*

Automation

Automating the compositionality rules. As discussed in Chapter 6, Hypra automates verification by leveraging HHL's core and loop rules. However, it does not yet automate the compositionality rules from Section 5.7,[9] which means that examples requiring these rules cannot be verified by Hypra. A natural next step is to develop methods for automating compositionality rules. One key challenge is to design a hint format that captures the essential insights needed for these proofs, while still enabling automation.

9: Except for the rule FrameSafe, as discussed in Section 6.4.3.

An alternative SMT-based encoding based on predicate transformers. Although our evaluation shows that Hypra performs well on benchmarks from the literature, its scalability to larger programs remains unclear, as these benchmarks are relatively small. One idea to improve performance is to use a more lightweight encoding based on the syntactic rules from Section 5.5, rather than tracking sets of states at each step. For a (loop-free) program and postcondition, we could syntactically compute the *weakest (hyper-)precondition* [259] and check whether the user-provided precondition entails it. However, a key challenge is that the weakest precondition for conditionals **if** (b) $\{C_1\}$ **else** $\{C_2\}$ cannot be expressed compositionally (that is, as a function of the weakest preconditions of C_1 and C_2) with our syntax for hyper-assertions.

[259]: Dijkstra (1975), *Guarded Commands, Nondeterminacy and Formal Derivation of Programs*

Hypertypes. Another promising direction is to combine verification in Hypra with a novel notion of *hypertypes*, *i.e.*, types for hyperproperties, which would allow for efficient, syntactic typing whenever possible. While type systems for secure information flow have a long history [208], extending them to cover a broader range of hyperproperties would require developing new types (and corresponding type systems) and

[208]: Volpano et al. (1996), *A Sound Type System for Secure Flow Analysis*

integrating them with SMT-based verification, since the typing rules may be too restrictive. A further benefit of this approach is the potential to reduce the number of user-provided hints; for example, typing rules for loops can eliminate the need for explicit loop invariants.

Automating inference of loop invariants. Automatically inferring loop invariants is another promising direction, as demonstrated by tools like Descartes [88] and ForEx [121]. For Hypra, this task is more complex because the quantification scheme is not fixed in advance, allowing invariants to combine different types of hyperproperties, and because Hypra can apply various rules. Addressing these challenges would significantly enhance automation.

[88]: Sousa et al. (2016), *Cartesian Hoare Logic for Verifying K-Safety Properties*

[121]: Beutner (2024), *Automated Software Verification of Hyperliveness*

Appendix A.

A.1. Small-Step Semantics of ParImp

The rules for the (small-step) operational semantics used in Chapter 2 and Chapter 3 are shown in Figure A.1 (for non-failing executions, denoted as $\langle C, (s,h)\rangle \rightarrow \langle C', (s',h')\rangle$) and Figure A.2 (for failing executions, denoted as $\langle C, (s,h)\rangle \rightarrow \bot$). These rules are standard and adapted from Vafeiadis [157]. Typing details are omitted to avoid clutter.

[157]: Vafeiadis (2011), *Concurrent Separation Logic and Operational Semantics*

To detect data races in the rules for failing executions (Figure A.2), we use the functions $accesses(C, s)$ and $writes(C, s)$, defined as follows.

Definition A.1.1 Auxiliary functions to detect data races.
The function $accesses(C, s)$ returns the set of locations read from (in the next execution step) by command C in a state with store s of local variables s:

$$
\begin{aligned}
&accesses(\texttt{skip}, s) && \triangleq \emptyset \\
&accesses(x := e, s) && \triangleq \emptyset \\
&accesses(x := r.v, s) && \triangleq \{s(r)\} \\
&accesses(r.v := e, s) && \triangleq \{s(r)\} \\
&accesses(r := \texttt{alloc}(e), s) && \triangleq \emptyset \\
&accesses(C_1; C_2, s) && \triangleq accesses(C_1, s) \\
&accesses(C_1 \mid\mid C_2, s) && \triangleq accesses(C_1, s) \cup accesses(C_2, s) \\
&accesses(\texttt{if}\ (b)\ \{C_1\}\ \texttt{else}\ \{C_2\}, s) && \triangleq \emptyset \\
&accesses(\texttt{while}\ (e)\ \{C\}, s) && \triangleq \emptyset \\
&accesses(\texttt{free}(r), s) && \triangleq \{s(r)\}
\end{aligned}
$$

The function $writes(C, s)$ returns the set of locations written to (in the next execution step) by command C in a state with store s of local variables s:

$$
\begin{aligned}
&writes(\texttt{skip}, s) && \triangleq \emptyset \\
&writes(x := e, s) && \triangleq \emptyset \\
&writes(x := r.v, s) && \triangleq \emptyset \\
&writes(r.v := e, s) && \triangleq \{s(r)\} \\
&writes(r := \texttt{alloc}(e), s) && \triangleq \emptyset \\
&writes(C_1; C_2, s) && \triangleq writes(C_1, s) \\
&writes(C_1 \mid\mid C_2, s) && \triangleq writes(C_1, s) \cup writes(C_2, s) \\
&writes(\texttt{if}\ (b)\ \{C_1\}\ \texttt{else}\ \{C_2\}, s) && \triangleq \emptyset \\
&writes(\texttt{while}\ (e)\ \{C\}, s) && \triangleq \emptyset \\
&accesses(\texttt{free}(r), s) && \triangleq \{s(r)\}
\end{aligned}
$$

$$\textsc{Seq1}\quad \langle \textbf{skip}; C_2, (s,h)\rangle \to \langle C_2, (s,h)\rangle$$

$$\textsc{Seq2}\quad \frac{\langle C_1, (s,h)\rangle \to \langle C_1', (s',h')\rangle}{\langle C_1; C_2, (s,h)\rangle \to \langle C_1'; C_2, (s',h')\rangle}$$

$$\textsc{If1}\quad \frac{[\![b]\!](s)}{\langle \textbf{if}\ (b)\ \{C_1\}\ \textbf{else}\ \{C_2\}, (s,h)\rangle \to \langle C_1, (s,h)\rangle}$$

$$\textsc{If2}\quad \frac{\neg[\![b]\!](s)}{\langle \textbf{if}\ (b)\ \{C_1\}\ \textbf{else}\ \{C_2\}, (s,h)\rangle \to \langle C_2, (s,h)\rangle}$$

$$\textsc{Par1}\quad \frac{\langle C_1, (s,h)\rangle \to \langle C_1', (s',h')\rangle}{\langle C_1 \mid\mid C_2, (s,h)\rangle \to \langle C_1' \mid\mid C_2, (s',h')\rangle}$$

$$\textsc{Par2}\quad \frac{\langle C_2, (s,h)\rangle \to \langle C_2', (s',h')\rangle}{\langle C_1 \mid\mid C_2, (s,h)\rangle \to \langle C_1 \mid\mid C_2', (s',h')\rangle}$$

$$\textsc{Par3}\quad \langle \textbf{skip} \mid\mid \textbf{skip}, (s,h)\rangle \to \langle \textbf{skip}, (s,h)\rangle$$

$$\textsc{Loop}\quad \langle \textbf{while}\ (b)\ \{C\}, (s,h)\rangle \to \langle \textbf{if}\ (b)\ \{C; \textbf{while}\ (b)\ \{C\}\}\ \textbf{else}\ \{\textbf{skip}\}, (s,h)\rangle$$

$$\textsc{Assign}\quad \langle \mathtt{x} := \mathtt{e}, (s,h)\rangle \to \langle \textbf{skip}, (s[x \mapsto [\![e]\!](s)], h)\rangle$$

$$\textsc{Alloc}\quad \frac{(l,v) \notin dom(h)}{\langle \mathtt{r} := \textbf{alloc}(\mathtt{e}), (s,h)\rangle \to \langle \textbf{skip}, (s[r \mapsto l], h[(l,v) \mapsto [\![e]\!](s)])\rangle}$$

$$\textsc{Write}\quad \frac{(s(r),v) \in dom(h)}{\langle \mathtt{r.v} := \mathtt{e}, (s,h)\rangle \to \langle \textbf{skip}, (s, h[(s(r),v) \mapsto [\![e]\!](s)])\rangle}$$

$$\textsc{Read}\quad \frac{(s(r),v) \in dom(h)}{\langle \mathtt{x} := \mathtt{r.v}, (s,h)\rangle \to \langle \textbf{skip}, (s[x \mapsto h(s(r),v)], h)\rangle}$$

$$\textsc{Free}\quad \frac{(s(r),v) \in dom(h)}{\langle \textbf{free}(r), (s,h)\rangle \to \langle \textbf{skip}, (s, h[(r,v) \mapsto \bot])\rangle}$$

Figure A.1.: Small-step semantics rules for non-failing executions.

$$\textsc{SeqA}\quad \frac{\langle C_1, (s,h)\rangle \to \bot}{\langle C_1; C_2, (s,h)\rangle \to \bot}$$

$$\textsc{ParA1}\quad \frac{\langle C_1, (s,h)\rangle \to \bot}{\langle C_1 \mid\mid C_2, (s,h)\rangle \to \bot}$$

$$\textsc{ParA2}\quad \frac{\langle C_2, (s,h)\rangle \to \bot}{\langle C_1 \mid\mid C_2, (s,h)\rangle \to \bot}$$

$$\textsc{RaceA1}\quad \frac{accesses(C_1,s) \cap writes(C_2,s) \neq \emptyset}{\langle C_1 \mid\mid C_2, (s,h)\rangle \to \bot}$$

$$\textsc{RaceA2}\quad \frac{writes(C_1,s) \cap accesses(C_2,s) \neq \emptyset}{\langle C_1 \mid\mid C_2, (s,h)\rangle \to \bot}$$

$$\textsc{ReadA}\quad \frac{(s(r),v) \notin dom(h)}{\langle \mathtt{x} := \mathtt{r.v}, (s,h)\rangle \to \bot}$$

$$\textsc{WriteA}\quad \frac{(s(r),v) \notin dom(h)}{\langle \mathtt{r.v} := \mathtt{e}, (s,h)\rangle \to \bot}$$

$$\textsc{FreeA}\quad \frac{(s(r),v) \notin dom(h)}{\langle \textbf{free}(r), (s,h)\rangle \to \bot}$$

$$\textsc{ReadNullA}\quad \frac{s(r) = null}{\langle \mathtt{x} := \mathtt{r.v}, (s,h)\rangle \to \bot}$$

$$\textsc{WriteNullA}\quad \frac{s(r) = null}{\langle \mathtt{r.v} := \mathtt{e}, (s,h)\rangle \to \bot}$$

$$\textsc{FreeNullA}\quad \frac{s(r) = null}{\langle \textbf{free}(r), (s,h)\rangle \to \bot}$$

Figure A.2.: Small-step semantics rules for failing executions.

A.2. An Example of Unsound Magic Wand Packaging in Viper

As explained in Example 4.2.1 (Chapter 4), packaging the wand

$$w \triangleq \mathbf{acc}(x.f) * (x.f = y \vee x.f = z) \mathrel{-\!\!*} \mathbf{acc}(x.f) * \mathbf{acc}(x.f.g)$$

using the FIA leads to unsound reasoning: Starting in a state with permission to `x.f`, `y.g`, and `z.g`, we can prove the assertion $\mathbf{acc}(x.f) * (\mathbf{acc}(y.g) \vee \mathbf{acc}(z.g)) * w$. However, a correct footprint of w must either have some permission to `x.f`, or permission to *both* `y.g` and `z.g`. Therefore, $\mathbf{acc}(x.f) * (\mathbf{acc}(y.g) \vee \mathbf{acc}(z.g)) * w$ is actually equivalent to false.

Viper currently implements the FIA, and it is possible to exploit the unsoundness of the FIA when packaging the wand w to prove the postcondition **`false`**, as shown in Figure A.3. While Viper does not directly support disjunctions of accessibility predicates, we can observe in Figure A.3 that the assertion $w * \mathbf{acc}(x.f) * (\mathbf{acc}(y.g) \vee \mathbf{acc}(z.g))$ holds after packaging the wand w. This example relies on Viper's *permission introspection* feature: The expression **`perm(y.g)`** (for a reference `y` and a field `g`) yields the permission amount of `y.g` held by the current execution, not counting resources inside packaged wands.

```
1  field f: Ref
2  field g: Int
3
4  method main(x:Ref, y:Ref, z:Ref)
5     requires acc(x.f) && acc(y.g) && acc(z.g)
6  {
7     package acc(x.f) && (x.f == y || x.f == z) --* acc(x.f) && acc(x.f.g)
8     {
9        assert x.f == y ? acc(y.g) : acc(z.g)
10    }
11    assert (acc(x.f) && (x.f == y || x.f == z) --* acc(x.f) && acc(x.f.g))
12         && acc(x.f) && (perm(y.g) == write || perm(z.g) == write)
13    if (perm(y.g) == write) {
14       x.f := y
15    }
16    else {
17       x.f := z
18    }
19    apply acc(x.f) && (x.f == y || x.f == z) --* acc(x.f) && acc(x.f.g)
20    assert false
21 }
```

Figure A.3.: A small Viper program that illustrates how to prove false using the unsoundness of the FIA. This program relies on Viper's *permission introspection* feature, which allows to inspect the amount of permission to a heap location owned by the current execution: The expression **perm**(`y.g`) yields the permission amount of `y.g` held by the current execution, not counting resources inside packaged wands. The Viper symbols && and || represent the separation conjunction $*$ and the disjunction $\vee$, respectively.

The program shown in Figure A.3 is currently verified by Viper. Method `main` starts in a state with permission to `x.f`, `y.g`, and `z.g`. We then package the wand w (lines 7-10) using the FIA, and help the proof search with the assertion on line 9 (the hints to guide a package statement are called *proof scripts*, and we discuss them in the extended version of Chapter 4 [198]). After the package statement, we assert $w * \mathbf{acc}(x.f) * (\mathbf{acc}(y.g) \vee \mathbf{acc}(z.g))$ (lines 11-12), using permission introspection to express the disjunction.[1] Using this magic wand, we can derive an explicit contradiction. To do this, we assign `y` to `x.f` if the current execution owns `y.g`, and `z` otherwise (lines 13-18), using permission introspection. Finally, we apply the wand (line 19).

[198]: Dardinier et al. (2022), *Sound Automation of Magic Wands (Extended Version)*

1: Contrary to accessibility predicates (such as $\mathbf{acc}(y.g)$), Viper allows combining disjunctions with permission introspection.

Viper is able to prove false on line 20, because:

- Either the current execution owns `y.g`, in which case the permission of `z.g` was computed as the footprint of w. Thus, the current execution satisfies the assertion $w * \mathbf{acc}(x.f) * \mathbf{acc}(y.g)$. Applying the wand w with `x.f = y` effectively exchanges ownership of `x.f` with ownership of `y.g`, resulting in a state that owns `y.g` twice, which is thus an inconsistent state.
- Or the current execution does not own `y.g`, which means that the permission of `y.g` was computed as the footprint of w, and thus the execution satisfies the assertion $w * \mathbf{acc}(x.f) * \mathbf{acc}(z.g)$. In this case, assigning `z` to `x.f` and then applying the wand w leads to an inconsistent state, which owns `z.g` twice.

A.3. Expressiveness of Hyper Triples

In this section, we demonstrate the expressiveness of hyper-triples (and thus Hyper Hoare Logic) by showing how they can express the judgments of existing over- and underapproximating Hoare logics (Section A.3.1 and Section A.3.2) and enable reasoning about useful properties that go beyond over- and underapproximation (Section A.3.3). All theorems and propositions in this section have been proved in Isabelle/HOL.

A.3.1. Overapproximate Hoare Logics

The vast majority of existing Hoare logics prove the absence of bad (combinations of) program executions. To achieve this, they prove properties *for all* (combinations of) executions, that is, they overapproximate the set of possible (combinations of) executions. In this subsection, we discuss *overapproximate logics* that prove properties of single executions or of k executions (for a fixed number k), and show that Hyper Hoare Logic goes beyond them by also supporting properties of unboundedly or infinitely many executions.

Single executions

Classical *Hoare Logic* [8, 9] is an overapproximate logic for properties of single executions. The meaning of triples can be defined as follows:

[8]: Floyd (1967), *Assigning Meanings to Programs*
[9]: Hoare (1969), *An Axiomatic Basis for Computer Programming*

Definition A.3.1 Hoare Logic (HL)**.**
Let P and Q be sets of extended states. Then

$$\models_{HL} \{P\}\ C\ \{Q\} \triangleq (\forall \varphi \in P.\, \forall \sigma'.\, \langle C, \varphi^P \rangle \to \sigma' \Rightarrow (\varphi^L, \sigma') \in Q)$$

This definition reflects the standard partial-correctness meaning of Hoare triples: executing C in some initial state that satisfies P can only lead to a final state that satisfies Q. This meaning can be expressed as a program hyperproperty as defined in Definition 5.3.6:

Proposition A.3.1 HL triples express hyperproperties**.**
Given sets of extended states P and Q, there exists a hyperproperty $\mathcal{H}$ such that, for all commands C, $C \in \mathcal{H}$ iff $\models_{HL} \{P\}\ C\ \{Q\}$.

Proof. We define

$$\mathcal{H} \triangleq \{C \mid \forall \varphi \in P.\, \forall \sigma'.\, (\varphi^P, \sigma') \in \Sigma(C) \Rightarrow (\varphi^L, \sigma') \in Q\}$$

and prove $\forall C.\, C \in \mathcal{H} \iff \models_{HL} \{P\}\ C\ \{Q\}$. □

This proposition together with completeness of Hyper Hoare Logic (Theorem 5.4.2) implies the *existence* of a proof in Hyper Hoare Logic for every valid classical Hoare triple. But there is an even stronger connection: we can map any assertion in classical Hoare logic to a hyper-assertion in Hyper Hoare Logic, which suggests a direct translation from classical Hoare logic to Hyper Hoare Logic.

The assertions P and Q of a valid Hoare triple characterize *all* initial and *all* final states of executing a command C. Consequently, they represent *upper bounds* on the possible initial and final states. We can use this observation to map classical Hoare triples to hyper-triples by interpreting their pre- and postconditions as upper bounds on sets of states.

Proposition A.3.2 Expressing HL in Hyper Hoare Logic.
Let $\overline{P} \triangleq (\lambda S.\, S \subseteq P)$.
Then $\models_{HL} \{P\}\ C\ \{Q\}$ *iff* $\models [\overline{P}]\ C\ [\overline{Q}]$.

Equivalently, $\models_{HL} \{P\}\ C\ \{Q\}$ *iff* $\models [\forall\langle\varphi\rangle.\, \varphi \in P]\ C\ [\forall\langle\varphi\rangle.\, \varphi \in Q]$.

This proposition implies that some rules of Hyper Hoare Logic have a direct correspondence in HL. For example, the rule *Seq* instantiated with $\overline{P}$, $\overline{R}$, and $\overline{Q}$ directly corresponds to the sequential composition rule from HL. Moreover, the upper-bound operator distributes over $\otimes$ and $\bigotimes$, since $\overline{A} \otimes \overline{B} = \overline{A \cup B}$, and $\bigotimes_i \overline{F_i} = \overline{\bigcup_i F(i)}$. Consequently, we can for example easily derive in Hyper Hoare Logic the classic while-rule from HL, using the rule *Iter* from Figure 5.2. Moreover, as HL can be encoded using syntactic hyper-assertions, we can also derive HL's rules for assignments and assume statements from HHl's syntactic rules (Section 5.5).

k executions

Many extensions of HL have been proposed to deal with hyperproperties of k executions. As a representative of this class of logics, we relate *Cartesian Hoare Logic* [88] to Hyper Hoare Logic. To define the meaning of Cartesian Hoare Logic triples, we first lift our semantic relation $\rightarrow$ from one execution on states to k executions on extended states.

[88]: Sousa et al. (2016), *Cartesian Hoare Logic for Verifying K-Safety Properties*

Definition A.3.2 Big-step semantics for k executions.
Let $k \in \mathbb{N}$. *We write*

- $\vec{\varphi}$ *to represent the k-tuple of extended states* $(\varphi_1, \ldots, \varphi_k)$,
- $\forall\vec{\varphi}$ *to represent* $\forall\varphi_1, \ldots, \varphi_k$,
- $\exists\vec{\varphi}$ *to represent* $\exists\varphi_1, \ldots, \varphi_k$,
- $\forall\langle\vec{\varphi}\rangle$ *to represent* $\forall\langle\varphi_1\rangle, \ldots, \langle\varphi_k\rangle$,
- $\exists\langle\vec{\varphi}\rangle$ *to represent* $\exists\langle\varphi_1\rangle, \ldots, \langle\varphi_k\rangle$,

Moreover, we define the relation $\overset{k}{\rightarrow}$ *as*

$$\langle C, \vec{\varphi}\rangle \overset{k}{\rightarrow} \vec{\varphi'} \triangleq (\forall i \in [1,k].\, \langle C, {\varphi_i}^P\rangle \rightarrow {\varphi'_i}^P \ \wedge\ {\varphi_i}^L = {\varphi'_i}^L)$$

Definition A.3.3 Cartesian Hoare Logic (CHL).
Let $k \in \mathbb{N}$, *and let* P *and* Q *be sets of k-tuples of extended states. Then*

$$\models_{CHL(k)} \{P\}\ C\ \{Q\} \triangleq (\forall\vec{\varphi} \in P.\, \forall\vec{\varphi'}.\, \langle C, \vec{\varphi}\rangle \overset{k}{\rightarrow} \vec{\varphi'} \Rightarrow \vec{\varphi'} \in Q)$$

$\models_{CHL(k)} \{P\}\ C\ \{Q\}$ is valid iff executing C k times in k initial states that *together* satisfy P can only lead to k final states that together satisfy Q. This meaning can be expressed as a program hyperproperty:

Proposition A.3.3 CHL triples express hyperproperties.
Given sets of k-tuples of extended states P and Q, there exists a hyperproperty $\mathcal{H}$ such that, for all commands C, $C \in \mathcal{H} \iff \models_{CHL(k)} \{P\}\ C\ \{Q\}$.

Proof. We define

$$\mathcal{H} \triangleq \{C \mid \forall \vec{\varphi} \in P.\, \forall \vec{\varphi'}.\, (\forall i \in [1,k].\, \varphi_i^L = \varphi_i'^L \wedge (\varphi_i^P, \varphi_i'^P) \in \Sigma(C)) \Rightarrow \vec{\varphi'} \in Q\}$$

and prove $\forall C.\, C \in \mathcal{H} \iff \models_{CHL(k)} \{P\}\ C\ \{Q\}$. □

Like we did for Hoare Logic, we can provide a direct translation from CHL triples to hyper-triples in our logic. Similarly to HL, CHL assertions express upper bounds, here on sets of k-tuples. However, simply using upper bounds as in Proposition A.3.2 does not capture the full expressiveness of CHL because executions in CHL are *distinguishable*. For example, one can express monotonicity from x to y as $\models_{CHL(k)} \{x(1) \geq x(2)\}\ y := x\ \{y(1) \geq y(2)\}$. When going from (ordered) tuples of states in CHL to (unordered) sets of states in Hyper Hoare Logic, we need to identify which state in the set of final states corresponds to execution 1, and which state corresponds to execution 2. As we did in Section 5.7.2 to express monotonicity, we use a logical variable t to tag a state with the number i of the execution it corresponds to.

Proposition A.3.4 Expressing CHL in Hyper Hoare Logic.
Let

$$P' \triangleq (\forall \langle \vec{\varphi} \rangle.\, \varphi_1^L(t) = 1 \Rightarrow \ldots \Rightarrow \varphi_k^L(t) = k \Rightarrow \vec{\varphi} \in P)$$
$$Q' \triangleq (\forall \langle \vec{\varphi} \rangle.\, \varphi_1^L(t) = 1 \Rightarrow \ldots \Rightarrow \varphi_k^L(t) = k \Rightarrow \vec{\varphi} \in Q)$$

where t does not occur free in P or Q. Then

$$\models_{CHL(k)} \{P\}\ C\ \{Q\} \iff \models [P']\ C\ [Q']$$

As an example, we can express the CHL assertion $y(1) \geq y(2)$ as the hyper-assertion

$$\forall \langle \varphi_1 \rangle, \langle \varphi_2 \rangle.\, \varphi_1^L(t) = 1 \Rightarrow \varphi_2^L(t) = 2 \Rightarrow \varphi_1^P(y) \geq \varphi_2^P(y)$$

Such translations provide a direct way of representing CHL proofs in Hyper Hoare Logic.

CHL, like Hyper Hoare Logic, can reason about multiple executions of a single command C, which is sufficient for many practically-relevant hyperproperties such as non-interference or determinism. Other logics, such as *Relational Hoare Logic* [103], relate the executions of multiple (potentially different) commands, for instance, to prove program equivalence. In case these commands are all the same, triples of relational logics can be translated to Hyper Hoare Logic analogously to CHL. We explain how to encode relational properties relating different commands to Hyper Hoare Logic in Section A.3.3.

[103]: Benton (2004), *Simple Relational Correctness Proofs for Static Analyses and Program Transformations*

Unboundedly many executions

To the best of our knowledge, all existing overapproximate Hoare logics consider a fixed number k of executions. In contrast, Hyper Hoare Logic can reason about an unbounded number of executions, as illustrated in the extended version of Chapter 5 [207]. Moreover, since our hyper-assertions are functions of potentially-infinite sets of states, Hyper Hoare Logic can even express properties of infinitely-many executions, as we illustrate in Section A.3.3.

[207]: Dardinier et al. (2024), *Hyper Hoare Logic*

A.3.2. Underapproximate Hoare Logics

Several recent Hoare logics allow proving the *existence* of certain (combinations of) program executions, which is useful, for instance, to disprove a specification, that is, to demonstrate that a program definitely has a bug. These logics underapproximate the set of possible (combinations of) executions. In this subsection, we discuss two forms of *underapproximate logics*, *backward underapproximate* and *forward underapproximate*, and show that both can be expressed in Hyper Hoare Logic.

Backward underapproximation

Reverse Hoare Logic [106] and *Incorrectness Logic* [89] are both underapproximate logics. Reverse Hoare Logic is designed to reason about the reachability of good final states. Incorrectness Logic uses the same ideas to prove the presence of bugs in programs. We focus on Incorrectness Logic in the following, but our results also apply to Reverse Hoare Logic. Incorrectness Logic reasons about single program executions:

[106]: de Vries et al. (2011), *Reverse Hoare Logic*

[89]: O'Hearn (2019), *Incorrectness Logic*

Definition A.3.4 Incorrectness Logic (IL)**.**
Let P and Q be sets of extended states. Then

$$\models_{IL} \{P\}\ C\ \{Q\} \triangleq (\forall \varphi \in Q.\, \exists \sigma.\, (\varphi^L, \sigma) \in P \land \langle C, \sigma \rangle \to \varphi^P)$$

The meaning of IL triples is defined *backward* from the postcondition: any state that satisfies the postcondition Q can be reached by executing C in an initial state that satisfies the precondition P. This meaning can be expressed as a program hyperproperty:

Proposition A.3.5 IL triples express hyperproperties**.**
Given sets of extended states P and Q, there exists a hyperproperty $\mathcal{H}$ such that, for all commands C, $C \in \mathcal{H}$ iff $\models_{IL} \{P\}\ C\ \{Q\}$.

Proof. We define

$$\mathcal{H} \triangleq \{C \mid \forall \varphi \in Q.\, \exists \sigma.\, (\varphi^L, \sigma) \in P \land (\sigma, \varphi^P) \in \Sigma(C)\}$$

and prove $\forall C.\, C \in \mathcal{H} \iff \models_{IL} \{P\}\ C\ \{Q\}$. □

Hoare Logic shows the absence of executions by overapproximating the set of possible executions, whereas Incorrectness Logic shows the existence of executions by underapproximating it. This duality also leads to an analogous translation of IL judgments into Hyper Hoare Logic, which uses lower bounds on the set of executions instead of the upper bounds used in Proposition A.3.2.

Proposition A.3.6 Expressing IL in Hyper Hoare Logic.
Let $\underline{P} \triangleq (\lambda S.\, P \subseteq S)$. *Then* $\models_{IL} \{P\}\ C\ \{Q\}$ *iff* $\models [\underline{P}]\ C\ [\underline{Q}]$.

Equivalently, $\models_{IL} \{P\}\ C\ \{Q\}$ *iff* $\models [\forall \varphi \in P.\, \langle \varphi \rangle]\ C\ [\forall \varphi \in Q.\, \langle \varphi \rangle]$.

Analogous to the upper bounds for HL, the lower-bound operator distributes over $\otimes$ and $\bigotimes$: $\underline{A} \otimes \underline{B} = \underline{A \cup B}$ and $\bigotimes_i \underline{F_i} = \underline{\bigcup_i F(i)}$. Using the latter equality with the rules *While* and *Cons*, it is easy to derive the loop rules from both Incorrectness Logic and Reverse Hoare Logic.

Murray [107] has recently proposed an underapproximate logic based on IL that can reason about two executions of two (potentially different) programs, for instance, to prove that a program violates a hyperproperty such as non-interference. We use the name 2-*Incorrectness Logic* for the restricted version of this logic where the two programs are the same (and discuss relational properties between different programs in Section A.3.3). The meaning of triples in k-Incorrectness Logic is also defined backward. They express that, for any pair of final states (φ_1', φ_2') that together satisfy a relational postcondition, there exist two initial states φ_1 and φ_2 that together satisfy the relational precondition, and executing command C in φ_1 (resp. φ_2) leads to φ_1' (resp. φ_2'). Our formalization lifts this meaning from 2 to k executions (and thus to *k-Incorrectness Logic*):

[107]: Murray (2020), *An Under-Approximate Relational Logic*

Definition A.3.5 k-Incorrectness Logic (k-IL).
Let $k \in \mathbb{N}$, *and* P *and* Q *be sets of* k*-tuples of extended states. Then*

$$\models_{k\text{–}IL} \{P\}\ C\ \{Q\} \triangleq (\forall \vec{\varphi'} \in Q.\, \exists \vec{\varphi} \in P.\, \langle C, \vec{\varphi} \rangle \xrightarrow{k} \vec{\varphi'})$$

Again, this meaning is a hyperproperty:

Proposition A.3.7 k-IL triples express hyperproperties.
Given sets of k*-tuples of extended states* P *and* Q, *there exists a hyperproperty* $\mathcal{H}$ *such that, for all commands* C, $C \in \mathcal{H} \iff \models_{k\text{–}IL} \{P\}\ C\ \{Q\}$.

Proof. We define

$$\mathcal{H} \triangleq \{C \mid \forall \vec{\varphi'} \in Q.\, \exists \vec{\varphi'} \in P.\, (\forall i \in [1,k].\, \varphi_i^L = \varphi'^L_i \wedge (\varphi_i^P, \varphi'^P_i) \in \Sigma(C))\}$$

and prove $\forall C.\, C \in \mathcal{H} \iff \models_{k\text{–}IL} \{P\}\ C\ \{Q\}$. □

Together with Theorem 5.3.2, this implies that we can express any k-IL triple as hyper-triple in Hyper Hoare Logic. However, defining a direct translation of k-IL triples to hyper-triples is surprisingly tricky. In particular, it is *not* sufficient to apply the transformation from Proposition A.3.4, which uses a logical variable t to tag each state with the number of the execution it belongs to. This approach works for Cartesian Hoare Logic

because CHL and Hyper Hoare Logic are both forward logics (see Definition 5.3.5 and Definition A.3.3). Intuitively, this commonality allows us to identify corresponding tuples from the preconditions in the two logics and relate them to corresponding tuples in the postconditions.

However, since k-IL is a *backward* logic, the same approach is not sufficient to identify corresponding tuples. For two states φ_1' and φ_2' from the set of final states, we know through the tag variable t to which execution they belong, but not whether they originated from one tuple $(\varphi_1, \varphi_2) \in P$, or from two *unrelated* tuples.

To solve this problem, we use another logical variable u, which records the "identity" of the initial k-tuple that satisfies P. To avoid cardinality issues, we define the encoding under the assumption that P depends only on program variables. Consequently, there are at most $|PStates^k|$ such k-tuples, which we can represent as logical values if the cardinality of $LVals$ is at least the cardinality of $PStates^k$, as shown by the following result:

Proposition A.3.8 Expressing k-IL in Hyper Hoare Logic.
Let t, u be distinct variables in LVars and

$$\begin{aligned} P' \triangleq (\forall \overrightarrow{\varphi} \in P. \varphi_1^L(t) = 1 \Rightarrow \ldots \Rightarrow \varphi_k^L(t) = k \\ \Rightarrow (\exists v. \langle \varphi_1[u \mapsto v] \rangle \wedge \ldots \wedge \langle \varphi_k[u \mapsto v] \rangle)) \\ Q' \triangleq (\forall \overrightarrow{\varphi'} \in Q. \varphi_1^L(t) = 1 \Rightarrow \ldots \Rightarrow \varphi_k^L(t) = k \\ \Rightarrow (\exists v. \langle \varphi_1'[u \mapsto v] \rangle \wedge \ldots \wedge \langle \varphi_k'[u \mapsto v] \rangle)) \end{aligned}$$

If (1) P depends only on program variables, (2) the cardinality of $LVals$ is at least the cardinality of $PStates^k$, and (3) t, u do not occur free in P or Q, then $\models_{k\text{-}IL} \{P\}\ C\ \{Q\} \iff \models [P']\ C\ [Q']$.

This proposition provides a direct translation for some k-IL triples into hyper-triples. Those that cannot be translated directly can still be verified with Hyper Hoare Logic, according to Proposition A.3.7.

Forward underapproximation

Underapproximate logics can also be formulated in a forward way: Executing command C in any state that satisfies the precondition reaches at least one final state that satisfies the postcondition. *Forward underapproximation*, sometimes also called *Lisbon Logic*, has recently been explored in both *Outcome Logic* [102], a Hoare logic whose goal is to unify correctness (in the sense of classical Hoare logic) and incorrectness reasoning (in the sense of forward underapproximation) for single program executions, and in *Sufficient Incorrectness Logic* (SIL) [214].

[102]: Zilberstein et al. (2023), *Outcome Logic*

[214]: Ascari et al. (2024), *Sufficient Incorrectness Logic*

Forward underapproximation for single executions, which corresponds to SIL, can be formalized as follows:

Definition A.3.6 Sufficient Incorrectness Logic (SIL).
Let P and Q be sets of extended states. Then

$$\models_{SIL} \{P\}\ C\ \{Q\} \triangleq \left(\forall \varphi \in P. \exists \sigma'. \langle C, \varphi^P \rangle \rightarrow \sigma' \wedge (\varphi^L, \sigma') \in Q\right)$$

This meaning can be expressed in Hyper Hoare Logic as follows: If we execute C in a set of initial states that contains at least one state from P then the set of final states will contain at least one state in Q.

Proposition A.3.9 Expressing SIL in Hyper Hoare Logic.

$$\models_{SIL} \{P\}\ C\ \{Q\} \iff \models [\lambda S.\, P \cap S \neq \varnothing]\ C\ [\lambda S.\, Q \cap S \neq \varnothing]$$

Equivalently, $\models_{SIL} \{P\}\ C\ \{Q\}$ *iff* $\models [\exists \langle \varphi \rangle.\, \varphi \in P]\ C\ [\exists \langle \varphi \rangle.\, \varphi \in Q]$.

The precondition (resp. postcondition) states that the intersection between S and P (resp. Q) is non-empty. If instead it required that S is a *non-empty subset* of P (resp. Q), it would express the meaning of Outcome Logic triples, *i.e.*, the conjunction of classical Hoare Logic and forward underapproximation.

While SIL reasons about single executions only, it is straightforward to generalize forward underapproximation to multiple executions:

Definition A.3.7 k-Forward Underapproximation (k-FU).
Let $k \in \mathbb{N}$, and let P and Q be sets of k-tuples of extended states. Then

$$\models_{k\text{–}FU} \{P\}\ C\ \{Q\} \triangleq (\forall \vec{\varphi} \in P.\, \exists \vec{\varphi'} \in Q.\, \langle C, \vec{\varphi} \rangle \xrightarrow{k} \vec{\varphi'})$$

Again, this meaning can be expressed as a hyperproperty:

Proposition A.3.10 k-FU triples express hyperproperties.
Given sets of k-tuples of extended states P and Q, there exists a hyperproperty $\mathcal{H}$ such that, for all commands C, $C \in \mathcal{H} \iff \models_{k\text{–}FU} \{P\}\ C\ \{Q\}$.

Proof. We define

$$\mathcal{H} \triangleq \{C \mid \forall \vec{\varphi} \in P.\, \exists \vec{\varphi'} \in Q.\, (\forall i \in [1,k].\, \varphi_i^L = \varphi'^L_i \wedge (\varphi_i^P, \varphi'^P_i) \in \Sigma(C))\}$$

and prove $\forall C.\, C \in \mathcal{H} \iff \models_{k\text{–}FU} \{P\}\ C\ \{Q\}$. □

Since SIL corresponds exactly to k-FU for $k = 1$, this proposition applies also to SIL.

Because k-FU is *forward* underapproximate, we can use the tagging from Proposition A.3.4 to translate k-FU triples into hyper-triples. The following encoding intuitively corresponds to the precondition $(S_1 \times \ldots \times S_k) \cap P \neq \varnothing$ and the postcondition $(S_1 \times \ldots \times S_k) \cap Q \neq \varnothing$, where S_i corresponds to the set of states with $t = i$:

Proposition A.3.11 Expressing k-FU in Hyper Hoare Logic.
Let

$$P' \triangleq (\exists \langle \vec{\varphi} \rangle.\, \varphi_1^L(t) = 1 \wedge \ldots \wedge \varphi_k^L(t) = k \wedge \vec{\varphi} \in P)$$
$$Q' \triangleq (\exists \langle \vec{\varphi} \rangle.\, \varphi_1^L(t) = 1 \wedge \ldots \wedge \varphi_k^L(t) = k \wedge \vec{\varphi} \in Q)$$

If t does not occur free in P or Q, then

$$\models_{k\text{–}FU} \{P\}\ C\ \{Q\} \iff \models [P']\ C\ [Q']$$

A.3.3. Beyond Over- and Underapproximation

In the previous subsections, we have discussed overapproximate logics, which reason about *all* executions, and underapproximate logics, which reason about the *existence* of executions. In this subsection, we explore program hyperproperties that combine universal and existential quantification, as well as properties that apply other comprehensions to the set of executions. We also discuss relational properties about multiple programs (such as program equivalence).

$\forall^*\exists^*$-hyperproperties

Generalized non-interference (see Section 5.2.3) intuitively expresses that for each execution that produces a given observable output, there exists another execution that produces the same output using any other secret. That is, observing the output does not reveal any information about the secret. GNI is a hyperproperty that cannot be expressed in existing over- or underapproximate Hoare logics. It mandates the existence of an execution *based on other possible executions*, whereas underapproximate logics can show only the existence of (combinations of) executions that satisfy some properties, *independently of the other possible executions*. Generalized non-interference belongs to a broader class of $\forall^*\exists^*$-hyperproperties.

RHLE [96] is a Hoare-style relational logic that has been recently proposed to verify $\forall^*\exists^*$-relational properties, such as program refinement [240]. We call the special case of RHLE where triples specify properties of multiple executions of the same command *k-Universal Existential*; we can formalize its triples as follows:

[96]: Dickerson et al. (2022), *RHLE*

[240]: Abadi et al. (1991), *The Existence of Refinement Mappings*

Definition A.3.8 k-Universal Existential (k-UE)**.**
Let $k_1, k_2 \in \mathbb{N}$, and let P and Q be sets of $(k_1 + k_2)$-tuples of extended states. Then

$$\begin{aligned}\models_{k\text{–}UE(k_1,k_2)} \{P\}\ C\ \{Q\} \triangleq (\forall(\vec{\varphi}, \vec{\gamma}) \in P.\, \forall\vec{\varphi'}.\, \langle C, \vec{\varphi}\rangle \xrightarrow{k_1} \vec{\varphi'} \\ \Rightarrow (\exists\vec{\gamma'}.\, \langle C, \vec{\gamma}\rangle \xrightarrow{k_2} \vec{\gamma'} \wedge (\vec{\varphi'}, \vec{\gamma'}) \in Q))\end{aligned}$$

Given $k_1 + k_2$ initial states $\varphi_1, \ldots, \varphi_{k_1}$ and $\gamma_1, \ldots, \gamma_{k_2}$ that together satisfy the precondition P, for any final states $\varphi'_1, \ldots, \varphi'_{k_1}$ that can be reached by executing C in the initial states $\varphi_1, \ldots, \varphi_{k_1}$, there exist k_2 final states $\gamma'_1, \ldots, \gamma'_{k_2}$ that can be reached by executing C in the initial states $\gamma_1, \ldots, \gamma_{k_2}$, such that $\varphi'_1, \ldots, \varphi'_{k_1}, \gamma'_1, \ldots, \gamma'_{k_2}$ together satisfy the postcondition Q.

The properties expressed by k-UE assertions are hyperproperties:

Proposition A.3.12 k-UE triples express hyperproperties**.**
Given sets of $(k_1 + k_2)$-tuples of extended states P and Q, there exists a

hyperproperty $\mathcal{H}$ such that, for all commands C,

$$C \in \mathcal{H} \iff \models_{k\text{-}UE(k_1,k_2)} \{P\}\ C\ \{Q\}$$

Proof. We define

$$\mathcal{H} \triangleq \{C \mid \forall(\vec{\varphi}, \vec{\gamma}) \in P.\, \forall\vec{\varphi'}.\, \Big(\forall i \in [1,k_1].\, (\varphi_i^P, \varphi'^P_i) \in \Sigma(C) \land \varphi_i^L = \varphi'^L_i\Big)$$
$$\Rightarrow \exists\vec{\gamma'}.\, (\vec{\varphi'}, \vec{\gamma'}) \in Q \land (\forall i \in [1,k_2].\, (\gamma_i^P, \gamma'^P_i) \in \Sigma(C) \land \gamma_i^L = \gamma'^L_i)\}$$

and prove $\forall C.\, C \in \mathcal{H} \iff \models_{k\text{-}UE(k_1,k_2)} \{P\}\ C\ \{Q\}$. □

They can be directly expressed in Hyper Hoare Logic, as follows:

Proposition A.3.13 Expressing k-UE in Hyper Hoare Logic.
Let t, u be distinct variables in LVars, and

$$T_n \triangleq (\lambda\vec{\varphi}.\, \forall i \in [1,k_n].\, \langle\varphi_i\rangle \land \varphi_i(t) = i \land \varphi_i(u) = n)$$
$$P' \triangleq (\forall i.\, \exists\langle\varphi\rangle.\, \varphi^L(t) = i \land \varphi^L(u) = 2)$$
$$\land (\forall\vec{\varphi}, \vec{\gamma}.\, T_1(\vec{\varphi}) \land T_2(\vec{\gamma}) \Rightarrow (\vec{\varphi}, \vec{\gamma}) \in P)$$
$$Q' \triangleq (\forall\vec{\varphi'}.\, T_1(\varphi') \Rightarrow (\exists\vec{\gamma'}.\, T_2(\vec{\gamma'}) \land (\vec{\varphi'}, \vec{\gamma'}) \in Q))$$

where t, u do not occur free in P or Q. Then

$$\models_{k\text{-}UE(k_1,k_2)} \{P\}\ C\ \{Q\} \iff \models [P']\ C\ [Q']$$

This proposition borrows ideas from the translations of other logics we saw earlier. In particular, we use a logical variable t to tag the executions, and an additional logical variable u that indicates whether a state is universally ($u = 1$) or existentially ($u = 2$) quantified.

$\exists^*\forall^*$-hyperproperties

To the best of our knowledge, no existing Hoare logic can express $\exists^*\forall^*$-hyperproperties, *i.e.*, the *existence* of executions in relation to *all* other executions. As shown by the example in Section 5.5, $\exists^*\forall^*$-hyperproperties naturally arise when disproving a $\forall^*\exists^*$-hyperproperty (such as GNI), where the existential part can be thought of as a counter-example, and the universal part as the proof that this is indeed a counter-example. The existence of a minimum for a function computed by a command C is another simple example of an $\exists^*\forall^*$-property, as illustrated in Section 5.6.3.

Properties using other comprehensions

Some interesting program hyperproperties cannot be expressed by quantifying over states, but require other comprehensions over the set of states, such as counting or summation. As an example, the hyperproperty "there are exactly n different possible outputs for any given input" cannot be expressed by quantifying over the states, but requires counting (see the extended version of Chapter 5 [207]). Other examples of such hyperproperties include statistical properties about a program:

[207]: Dardinier et al. (2024), *Hyper Hoare Logic*

Example A.3.1 Mean number of requests.
Consider a command C that, given some input x, retrieves and returns information from a database. At the end of the execution of C, variable n contains the number of database requests that were performed. If the input values are uniformly distributed (and with a suitable precondition P), then the following hyper-triple expresses that the average number of requests performed by C is at most 2:

$$[P]\ C\ [\lambda S.\, mean_n^x(\{\varphi^P \mid \varphi \in S\}) \leq 2]$$

where $mean_n^x$ computes the average (using a suitable definition for the average if the set is infinite) of the value of n.

To the best of our knowledge, Hyper Hoare Logic is the only Hoare logic that can prove this property; existing logics neither support reasoning about mean-comprehensions over multiple execution states nor reasoning about infinitely many executions *at the same time* (which is necessary if the domain of input x is infinite).

Relational program properties

Relational program properties typically relate executions of several *different* programs and, thus, do not correspond to program hyperproperties as defined in Definition 5.3.6. However, it is possible to construct a single *product* program [123, 125] that encodes the executions of several given programs, such that relational properties can be expressed as hyperproperties of the constructed program and proved in Hyper Hoare Logic.

[123]: Barthe et al. (2011), *Relational Verification Using Product Programs*
[125]: Barthe et al. (2013), *Beyond 2-Safety*

We illustrate this approach on *program refinement* [240]. A command C_2 *refines* a command C_1 iff the set of pairs of pre- and post-states of C_2 is a subset of the corresponding set of C_1. Program refinement is a $\forall\exists$-property, where the $\forall$ and the $\exists$ apply to different programs. To encode refinement, we construct a new product program that non-deterministically executes either C_1 or C_2, and we track in a logical variable t which command was executed. This encoding allows us to express and prove refinement in Hyper Hoare Logic:

[240]: Abadi et al. (1991), *The Existence of Refinement Mappings*

Example A.3.2 Expressing program refinement in Hyper Hoare Logic.
Let $C \triangleq (t := 1; C_1) + (t := 2; C_2)$. If t does not occur free in C_1 or C_2 then C_2 refines C_1 iff

$$\models [\lambda S.\, |S| = 1]\ C\ [\forall \langle \varphi \rangle.\, \varphi^P(t) = 2 \Rightarrow \langle (\varphi^L, \varphi^P[t := 1]) \rangle]$$

This example illustrates the general methodology to transform a relational property over different programs into an equivalent hyperproperty for a new product program, and thus to reason about relational program properties in Hyper Hoare Logic. Relational logics typically provide rules that align and relate parts of the different program executions; we present such a rule for Hyper Hoare Logic in the extended version of Chapter 5 [207].

[207]: Dardinier et al. (2024), *Hyper Hoare Logic*

This section demonstrated that hyper-triples (and thus HHL) is sufficiently expressive to prove and disprove arbitrary program hyperproperties as defined in Definition 5.3.6. Thereby, it captures hyperproperties that are beyond the reach of existing Hoare logics.

A.4. Examples of Derivations in HHL

A.4.1. Monotonicity of the Fibonacci Sequence

In this subsection, we show the proof that the program C_{fib}, described in Example 5.2.3, is monotonic. Precisely, we prove the triple

$$\frac{[\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \varphi_1(t){=}1 \wedge \varphi_2(t){=}2 \Rightarrow \varphi_1(n) \geq \varphi_2(n)]}{C_{fib}}{[\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \varphi_1(t){=}1 \wedge \varphi_2(t){=}2 \Rightarrow \varphi_1(a) \geq \varphi_2(a)]}$$

using the rule While-$\forall^*\exists^*$ with the loop invariant

$$I \triangleq \square(b \geq a \geq 0) \wedge (\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \varphi_1(t){=}1 \wedge \varphi_2(t){=}2 \Rightarrow (\varphi_1(n){-}\varphi_1(i) \geq \varphi_2(n){-}\varphi_2(i) \wedge \varphi_1(a) \geq \varphi_2(a) \wedge \varphi_1(b) \geq \varphi_2(b)))$$

$\{\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \varphi_1(t){=}1 \wedge \varphi_2(t){=}2 \Rightarrow \varphi_1(n) \geq \varphi_2(n)\}$

$\{(\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \varphi_1(t){=}1 \wedge \varphi_2(t){=}2 \Rightarrow (\varphi_1(n){-}0 \geq \varphi_2(n){-}0 \wedge 0 \geq 0 \wedge 1 \geq 1)) \wedge \square(1 \geq a \geq 0)\}$ (Cons)

$a := 0;$

$\{(\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \varphi_1(t){=}1 \wedge \varphi_2(t){=}2 \Rightarrow (\varphi_1(n){-}0 \geq \varphi_2(n){-}0 \wedge \varphi_1(a) \geq \varphi_2(a) \wedge 1 \geq 1)) \wedge \square(1 \geq a \geq 0)\}$ (AssignS)

$b := 1;$

$\{(\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \varphi_1(t){=}1 \wedge \varphi_2(t){=}2 \Rightarrow (\varphi_1(n){-}0 \geq \varphi_2(n){-}0 \wedge \varphi_1(a) \geq \varphi_2(a) \wedge \varphi_1(b) \geq \varphi_2(b))) \wedge \square(b \geq a \geq 0)\}$ (AssignS)

$i := 0;$

$\{(\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \varphi_1(t){=}1 \wedge \varphi_2(t){=}2 \Rightarrow (\varphi_1(n){-}\varphi_1(i) \geq \varphi_2(n){-}\varphi_2(i) \wedge \varphi_1(a) \geq \varphi_2(a) \wedge \varphi_1(b) \geq \varphi_2(b))) \wedge \square(b \geq a \geq 0)\}$ (AssignS)

Figure A.4.: First part of the proof, which proves that the loop invariant I holds before the loop.

Figure A.4 shows the (trivial) first part of the proof, which proves that the loop invariant I holds before the loop, and Figure A.5 shows the proof of $\vdash [I]$ **if** $(i < n)$ $\{C_{body}\}$ $[I]$, the first premise of the rule While-$\forall^*\exists^*$ (the second premise is trivial). In Figure A.5, we first record the initial values of a, b, and i in the logical variables v_a, v_b, and v_i, respectively, using the rule LUpdateS presented in Section 5.7. We then split our new hyper-assertion into a simple part, $\forall\langle\varphi\rangle.\ \varphi(i) = \varphi(v_i) \wedge \varphi(a) = \varphi(v_a) \wedge \varphi(b) = \varphi(v_b)$, and a frame F which stores the relevant information from the invariant I with the initial values. The hyper-assertion F is then framed around the if-statement, using the rule FrameSafe from Section 5.7. The proof of each branch is straightforward; the postconditions of the two branches are combined via the rule Choice.

We finally conclude with the rule Choice. This last entailment is justified by a case distinction. Let φ_1, φ_2 be two states such that $\varphi_1(t) = 1$, $\varphi_2(t) = 2$, and $\langle\varphi_1\rangle$ and $\langle\varphi_2\rangle$ hold. From the frame F, we know that $\varphi_1(v_a) \geq \varphi_2(v_a)$, and $\varphi_1(v_b) \geq \varphi_2(v_b)$. We conclude the proof by distinguishing the following three cases (the proof for each case is straightforward):

1. Both φ_1 and φ_2 took the first branch of the if statement, *i.e.*, $\varphi_1(v_i) < \varphi_1(n)$ and $\varphi_2(v_i) < \varphi_2(n)$, and thus both are in the set characterized by Q_1.
2. Both φ_1 and φ_2 took the second branch, *i.e.*, $\varphi_1(v_i) \geq \varphi_1(n)$ and $\varphi_2(v_i) \geq \varphi_2(n)$, and thus both are in the set characterized by Q_2.

$\{\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \varphi_1(t){=}1\wedge\varphi_2(t){=}2 \Rightarrow (\varphi_1(n){-}\varphi_1(i) \geq \varphi_2(n){-}\varphi_2(i) \wedge \varphi_1(a) \geq \varphi_2(a) \wedge \varphi_1(b) \geq \varphi_2(b)) \wedge \Box(b \geq a \geq 0)\}$

$\{\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \varphi_1(t){=}1\wedge\varphi_2(t){=}2 \Rightarrow (\varphi_1(n){-}\varphi_1(i) \geq \varphi_2(n){-}\varphi_2(i) \wedge \varphi_1(a) \geq \varphi_2(a) \wedge \varphi_1(b) \geq \varphi_2(b)) \wedge \Box(b \geq a \geq 0) \wedge \Box(v_a = a \wedge v_b = b \wedge v_i = i)\}$ (LUpdateS)

$\{(\forall\langle\varphi\rangle.\ \varphi(i) = \varphi(v_i) \wedge \varphi(a) = \varphi(v_a) \wedge \varphi(b) = \varphi(v_b))$
$\wedge \underbrace{(\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \varphi_1(t) = 1 \wedge \varphi_2(t) = 2 \Rightarrow \varphi_1(v_a) \geq \varphi_2(v_a) \geq 0 \wedge \varphi_1(v_b) \geq \varphi_2(v_b) \geq 0 \wedge \varphi_1(n) - \varphi_1(v_i) \geq \varphi_2(n) - \varphi_2(v_i))}_{F}\}$ (Cons)

$\{\forall\langle\varphi\rangle.\ \varphi(i) = \varphi(v_i) \wedge \varphi(a) = \varphi(v_a) \wedge \varphi(b) = \varphi(v_b)\}$
if $(*)$ {
$\{\forall\langle\varphi\rangle.\ \varphi(i) = \varphi(v_i) \wedge \varphi(a) = \varphi(v_a) \wedge \varphi(b) = \varphi(v_b)\}$
$\{\forall\langle\varphi\rangle.\ \varphi(i) < \varphi(n) \Rightarrow \varphi(v_i) < \varphi(n) \wedge \varphi(i) + 1 = \varphi(v_i) + 1 \wedge \varphi(b) = \varphi(v_b) \wedge \varphi(a) + \varphi(b) = \varphi(v_a) + \varphi(v_b)\}$ (Cons)
assume $i < n$;
$\{\forall\langle\varphi\rangle.\ \varphi(v_i) < \varphi(n) \wedge \varphi(i) + 1 = \varphi(v_i) + 1 \wedge \varphi(b) = \varphi(v_b) \wedge \varphi(a) + \varphi(b) = \varphi(v_a) + \varphi(v_b)\}$ (AssumeS)
$tmp := b$;
$b := a + b$;
$a := tmp$;
$i := i + 1$
$\underbrace{\{\forall\langle\varphi\rangle.\ \varphi(v_i) < \varphi(n) \wedge \varphi(i) = \varphi(v_i) + 1 \wedge \varphi(a) = \varphi(v_b) \wedge \varphi(b) = \varphi(v_a) + \varphi(v_b)\}}_{Q_1}$ (AssignS)
}
else {
$\{\forall\langle\varphi\rangle.\ \varphi(i) = \varphi(v_i) \wedge \varphi(a) = \varphi(v_a) \wedge \varphi(b) = \varphi(v_b)\}$
$\{\forall\langle\varphi\rangle.\ \varphi(i) \geq \varphi(n) \Rightarrow \varphi(v_i) \geq \varphi(n) \wedge \varphi(i) = \varphi(v_i) \wedge \varphi(a) = \varphi(v_a) \wedge \varphi(b) = \varphi(v_b)\}$ (Cons)
assume $\neg(i < n)$
$\underbrace{\{\forall\langle\varphi\rangle.\ \varphi(v_i) \geq \varphi(n) \wedge \varphi(i) = \varphi(v_i) \wedge \varphi(a) = \varphi(v_a) \wedge \varphi(b) = \varphi(v_b)\}}_{Q_2}$ (AssumeS)
}
$\{Q_1 \otimes Q_2\}$ (Choice)
$\{(Q_1 \otimes Q_2) \wedge F\}$ (FrameSafe)
$\{\forall\langle\varphi_1\rangle,\langle\varphi_2\rangle.\ \varphi_1(t){=}1\wedge\varphi_2(t){=}2 \Rightarrow (\varphi_1(n){-}\varphi_1(i) \geq \varphi_2(n){-}\varphi_2(i) \wedge \varphi_1(a) \geq \varphi_2(a) \wedge \varphi_1(b) \geq \varphi_2(b)) \wedge \Box(b \geq a \geq 0)\}$ (Cons)

Figure A.5.: Second part of the proof. This proof outline shows $\vdash [I]$ **if** $(i < n)$ $\{C_{body}\}$ $[I]$, the first premise of the rule *While*-$\forall^*\exists$, where C_{body} refers to the body of the loop.

3. φ_1 took the first branch and φ_2 took the second branch, *i.e.*, $\varphi_1(v_i) < \varphi_1(n)$ and $\varphi_2(v_i) \geq \varphi_2(n)$, and thus φ_1 is in the set characterized by Q_1 and φ_2 is in the set characterized by Q_2.

Importantly, the fourth case is not possible, because this would imply $\varphi_2(n) - \varphi_2(v_i) > 0 \geq \varphi_1(n) - \varphi_1(v_i)$, which contradicts the inequality $\varphi_1(n) - \varphi_1(v_i) \geq \varphi_2(n) - \varphi_2(v_i)$ from the frame F.

A.4.2. Existence of a Minimum

This subsection contains the proof, using the rule While-∃, that the program C_m from Example 5.6.5, satisfies the triple

$$[\neg emp \wedge \Box(k \geq 0)]\ C_m\ [\exists\langle\varphi\rangle.\, \forall\langle\alpha\rangle.\, \varphi(x) \leq \alpha(x) \wedge \varphi(y) \leq \alpha(y)]$$

Figure A.6 contains the (trivial) first part of the proof, which justifies that the hyper-assertion $\exists\langle\varphi\rangle.\, P_\varphi$, where

$$P_\varphi \triangleq (\forall\langle\alpha\rangle.\, 0 \leq \varphi(x) \leq \alpha(x) \wedge 0 \leq \varphi(y) \leq \alpha(y) \wedge \varphi(k) \leq \alpha(k) \wedge \varphi(i) = \alpha(i))$$

holds before the loop, as required by the precondition of the conclusion of the rule While-∃.

Figure A.7 shows the proof of the first premise of the rule While-∃, namely, for all v,

$$\begin{array}{c}[\exists\langle\varphi\rangle.\, P_\varphi \wedge \varphi(i) < \varphi(k) \wedge v = \varphi(k) - \varphi(i)] \\ \textbf{if}\ (i < k)\ \{C_{body}\} \\ [\exists\langle\varphi\rangle.\, P_\varphi \wedge \varphi(k) - \varphi(i) \prec v]\end{array}$$

where C_{body} is the body of the loop.

Finally, Figure A.8 shows the proof of the second premise of the rule While-∃. More precisely, it shows

$$\forall\varphi.\, \vdash [Q_\varphi]\ \textbf{if}\ (i < k)\ \{C_{body}\}\ [Q_\varphi]$$

where

$$Q_\varphi \triangleq \forall\langle\alpha\rangle.\, 0 \leq \varphi(x) \leq \alpha(x) \wedge 0 \leq \varphi(y) \leq \alpha(y)$$

from which we easily derive the second premise of the rule While-∃, using the rule Cons (since P_φ clearly entails Q_φ), and the rule While-∀*∃*.

$$\begin{array}{ll}
\{\neg emp \wedge \Box(k \geq 0)\} & \\
\{\exists\langle\varphi\rangle.\, \forall\langle\alpha\rangle.\, 0 \leq 0 \leq 0 \wedge 0 \leq 0 \leq 0 \wedge \varphi(k) \leq \alpha(k) \wedge 0 = 0\} & \text{(Cons)} \\
x := 0; & \\
\{\exists\langle\varphi\rangle.\, \forall\langle\alpha\rangle.\, 0 \leq \varphi(x) \leq \alpha(x) \wedge 0 \leq 0 \leq 0 \wedge \varphi(k) \leq \alpha(k) \wedge 0 = 0\} & \text{(AssignS)} \\
y := 0; & \\
\{\exists\langle\varphi\rangle.\, \forall\langle\alpha\rangle.\, 0 \leq \varphi(x) \leq \alpha(x) \wedge 0 \leq \varphi(y) \leq \alpha(y) \wedge \varphi(k) \leq \alpha(k) \wedge 0 = 0\} & \text{(AssignS)} \\
i := 0; & \\
\{\exists\langle\varphi\rangle.\, \forall\langle\alpha\rangle.\, 0 \leq \varphi(x) \leq \alpha(x) \wedge 0 \leq \varphi(y) \leq \alpha(y) \wedge \varphi(k) \leq \alpha(k) \wedge \varphi(i) = \alpha(i)\} & \text{(AssignS)}
\end{array}$$

Figure A.6.: First part of the proof: Proving the first loop invariant $\exists\langle\varphi\rangle.\, P_\varphi$.

$\{\exists\langle\varphi\rangle.(\forall\langle\alpha\rangle.0 \le \varphi(x) \le \alpha(x) \land 0 \le \varphi(y) \le \alpha(y) \land \varphi(k) \le \alpha(k) \land \varphi(i) = \alpha(i)) \land \varphi(i) < \varphi(k) \land v = \varphi(k) - \varphi(i)\}$

if $(i < k)$ {

$\{\exists\langle\varphi\rangle.(\forall\langle\alpha\rangle.0 \le \varphi(x) \le \alpha(x) \land 0 \le \varphi(y) \le \alpha(y) \land \varphi(k) \le \alpha(k) \land \varphi(i) = \alpha(i)) \land \varphi(i) < \varphi(k) \land v = \varphi(k) - \varphi(i) \land \square(i < k)\}$

$\{\exists\langle\varphi\rangle.\exists u.u \ge 2 \land (\forall\langle\alpha\rangle.\forall v.v \ge 2 \Rightarrow 0 \le 2 * \varphi(x) + u \le 2 * \alpha(x) + v \land 0 \le \varphi(y) + \varphi(x) * u \le \alpha(y) + \alpha(x) * v$
$\land \varphi(k) \le \alpha(k) \land \varphi(i) + 1 = \alpha(i) + 1) \land \varphi(k) - (\varphi(i)+1) < v\}$ (Cons (1))

$r \coloneqq nonDet();$

assume $r \ge 2;$

$\{\exists\langle\varphi\rangle.(\forall\langle\alpha\rangle.0 \le 2 * \varphi(x) + \varphi(r) \le 2 * \alpha(x) + \alpha(r) \land 0 \le \varphi(y) + \varphi(x) * \varphi(r) \le \alpha(y) + \alpha(x) * \alpha(r) \land \varphi(k) \le \alpha(k) \land \varphi(i) + 1 = \alpha(i) + 1)$
$\land \varphi(k) - (\varphi(i)+1) < v\}$ (HavocS, AssumeS)

$t \coloneqq x;$

$x \coloneqq 2 * x + r;$

$y \coloneqq y + t * r;$

$i \coloneqq i + 1$

$\{\exists\langle\varphi\rangle.(\forall\langle\alpha\rangle.0 \le \varphi(x) \le \alpha(x) \land 0 \le \varphi(y) \le \alpha(y) \land \varphi(k) \le \alpha(k) \land \varphi(i) = \alpha(i)) \land \varphi(k) - \varphi(i) < v\}$ (AssignS)

}

else {

$\{\exists\langle\varphi\rangle.(\forall\langle\alpha\rangle.0 \le \varphi(x) \le \alpha(x) \land 0 \le \varphi(y) \le \alpha(y) \land \varphi(k) \le \alpha(k) \land \varphi(i) = \alpha(i)) \land \varphi(i) < \varphi(k) \land v = \varphi(k) - \varphi(i) \land \square(i \ge k)\}$

$\{\exists\langle\varphi\rangle.(\forall\langle\alpha\rangle.0 \le \varphi(x) \le \alpha(x) \land 0 \le \varphi(y) \le \alpha(y) \land \varphi(k) \le \alpha(k) \land \varphi(i) = \alpha(i)) \land \varphi(k) - \varphi(i) < v\}$ (Cons (2))

skip

$\{\exists\langle\varphi\rangle.(\forall\langle\alpha\rangle.0 \le \varphi(x) \le \alpha(x) \land 0 \le \varphi(y) \le \alpha(y) \land \varphi(k) \le \alpha(k) \land \varphi(i) = \alpha(i)) \land \varphi(k) - \varphi(i) < v\}$ (Skip)

}

$\{\exists\langle\varphi\rangle.(\forall\langle\alpha\rangle.0 \le \varphi(x) \le \alpha(x) \land 0 \le \varphi(y) \le \alpha(y) \land \varphi(k) \le \alpha(k) \land \varphi(i) = \alpha(i)) \land \varphi(k) - \varphi(i) < v\}$ (IfSync)

Figure A.7.: Second part of the proof. Proving the first premise of the rule While-∃,

$$\forall v.\, \exists\langle\varphi\rangle.\, \vdash [P_\varphi \land \varphi(i) < \varphi(k) \land v = \varphi(k) - \varphi(i)]\ \textbf{if}\ (i < k)\ \{C_{body}\}\ [\exists\langle\varphi\rangle.\, P_\varphi \land \varphi(k) - \varphi(i) < v]$$

For Cons (1), we simply choose $u = 2$. For Cons (2), we notice that $\varphi(i) < \varphi(k)$ and $\square(i \ge k)$ are inconsistent (this branch is not taken at this stage), and thus the entailment trivially holds.

$\{\forall\langle\alpha\rangle.\, 0 \le \varphi(x) \le \alpha(x) \land 0 \le \varphi(y) \le \alpha(y)\}$

if $(*)$ {

$\{\forall\langle\alpha\rangle.\, 0 \le \varphi(x) \le \alpha(x) \land 0 \le \varphi(y) \le \alpha(y)\}$

$\{\forall\langle\alpha\rangle.\, \alpha(i) < \alpha(k) \Rightarrow \forall v.\, v \ge 2 \Rightarrow 0 \le \varphi(x) \le 2 * \alpha(x) + v \land 0 \le \varphi(y) \le \alpha(y) + \alpha(x) * v\}$ (Cons)

assume $i < k$;

$\{\forall\langle\alpha\rangle.\, \forall v.\, v \ge 2 \Rightarrow 0 \le \varphi(x) \le 2 * \alpha(x) + v \land 0 \le \varphi(y) \le \alpha(y) + \alpha(x) * v\}$ (AssumeS)

$r := nonDet()$;

$\{\forall\langle\alpha\rangle.\, \alpha(r) \ge 2 \Rightarrow 0 \le \varphi(x) \le 2 * \alpha(x) + \alpha(r) \land 0 \le \varphi(y) \le \alpha(y) + \alpha(x) * \alpha(r)\}$ (HavocS)

assume $r \ge 2$;

$\{\forall\langle\alpha\rangle.\, 0 \le \varphi(x) \le 2 * \alpha(x) + \alpha(r) \land 0 \le \varphi(y) \le \alpha(y) + \alpha(x) * \alpha(r)\}$ (AssumeS)

$t := x$;

$x := 2 * x + r$;

$y := y + t * r$;

$i := i + 1$

$\{\forall\langle\alpha\rangle.\, 0 \le \varphi(x) \le \alpha(x) \land 0 \le \varphi(y) \le \alpha(y)\}$ (AssignS)

}

else {

$\{\forall\langle\alpha\rangle.\, 0 \le \varphi(x) \le \alpha(x) \land 0 \le \varphi(y) \le \alpha(y)\}$

$\{\forall\langle\alpha\rangle.\, \alpha(i) \ge \alpha(k) \Rightarrow 0 \le \varphi(x) \le \alpha(x) \land 0 \le \varphi(y) \le \alpha(y)\}$ (Cons)

assume $i \ge k$;

skip

$\{\forall\langle\alpha\rangle.\, 0 \le \varphi(x) \le \alpha(x) \land 0 \le \varphi(y) \le \alpha(y)\}$ (AssumeS, Skip)

}

$\{(\forall\langle\alpha\rangle.\, 0 \le \varphi(x) \le \alpha(x) \land 0 \le \varphi(y) \le \alpha(y)) \otimes (\forall\langle\alpha\rangle.\, 0 \le \varphi(x) \le \alpha(x) \land 0 \le \varphi(y) \le \alpha(y))\}$ (Choice)

$\{\forall\langle\alpha\rangle.\, 0 \le \varphi(x) \le \alpha(x) \land 0 \le \varphi(y) \le \alpha(y)\}$ (Cons)

Figure A.8.: Third part of the proof. This proof outline shows $\forall\varphi.\, \vdash [Q_\varphi]$ **if** $(i < k)$ $\{C_{body}\}$ $[Q_\varphi]$.

Bibliography

[1] Dorota Huizinga and Adam Kolawa. *Automated Defect Prevention: Best Practices in Software Management*. Aug. 2007 (cited on page 1).

[2] Barton P. Miller, Lars Fredriksen, and Bryan So. 'An Empirical Study of the Reliability of UNIX Utilities'. In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. DOI: 10.1145/96267.96279 (cited on page 1).

[3] Koen Claessen and John Hughes. 'QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs'. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ICFP '00. New York, NY, USA: Association for Computing Machinery, Sept. 2000, pp. 268–279. DOI: 10.1145/351240.351266 (cited on page 1).

[4] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 'Symbolic Model Checking without BDDs'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by W. Rance Cleaveland. Berlin, Heidelberg: Springer, 1999, pp. 193–207. DOI: 10.1007/3-540-49059-0_14 (cited on page 1).

[5] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 'Bounded Model Checking Using Satisfiability Solving'. In: *Formal Methods in System Design* 19.1 (July 2001), pp. 7–34. DOI: 10.1023/A:1011276507260 (cited on page 1).

[6] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. 'Bounded Model Checking'. In: *Advances in Computers*. Vol. 58. Elsevier, Jan. 2003, pp. 117–148. DOI: 10.1016/S0065-2458(03)58003-2 (cited on page 1).

[7] Martin Leucker and Christian Schallhart. 'A Brief Account of Runtime Verification'. In: *The Journal of Logic and Algebraic Programming*. The 1st Workshop on Formal Languages and Analysis of Contract-Oriented Software (FLACOS'07) 78.5 (May 2009), pp. 293–303. DOI: 10.1016/j.jlap.2008.08.004 (cited on page 1).

[8] Robert W. Floyd. 'Assigning Meanings to Programs'. In: *Proceedings of Symposium in Applied Mathematics* (1967). Ed. by J. T. Schwartz, pp. 19–32 (cited on pages 1, 7, 119, 141, 187).

[9] C. A. R. Hoare. 'An Axiomatic Basis for Computer Programming'. In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. DOI: 10.1145/363235.363259 (cited on pages 1, 7, 119, 141, 187).

[10] J.C. Reynolds. 'Separation Logic: A Logic for Shared Mutable Data Structures'. In: *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*. July 2002, pp. 55–74. DOI: 10.1109/LICS.2002.1029817 (cited on pages 1–3, 5, 17, 136, 180).

[11] Peter W. O'Hearn. 'Resources, Concurrency, and Local Reasoning'. In: *Theoretical Computer Science*. Festschrift for John C. Reynolds's 70th Birthday 375.1 (May 2007), pp. 271–307. DOI: 10.1016/j.tcs.2006.12.035 (cited on pages 1–3, 10, 21, 24, 56, 67, 180).

[12] K. Rustan M. Leino. 'This Is Boogie 2'. In: (June 2008) (cited on pages 1, 8, 17, 102, 170, 172).

[13] K. Rustan M. Leino. 'Dafny: An Automatic Program Verifier for Functional Correctness'. In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Ed. by Edmund M. Clarke and Andrei Voronkov. Berlin, Heidelberg: Springer, 2010, pp. 348–370. DOI: 10.1007/978-3-642-17511-4_20 (cited on pages 1, 2, 8, 17, 172, 177, 179).

[14] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. 'Dependent Types and Multi-Monadic Effects in F*'. In: *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '16. New York, NY, USA: Association for Computing Machinery, Jan. 2016, pp. 256–270. DOI: 10.1145/2837614.2837655 (cited on pages 1, 49, 104, 179).

[15] 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*. Ed. by Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi. Berlin, Heidelberg: Springer, 2011, pp. 41–55. DOI: 10.1007/978-3-642-20398-5_4 (cited on pages 1, 2, 4, 5, 9, 10, 19, 21, 36, 37, 52, 67, 74, 86, 178).

[16] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 'Viper: A Verification Infrastructure for Permission-Based Reasoning'. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Barbara Jobstmann and K. Rustan M. Leino. Berlin, Heidelberg: Springer, 2016, pp. 41–62. DOI: 10.1007/978-3-662-49122-5_2 (cited on pages 1, 2, 4, 5, 8–10, 17, 21, 46, 52, 67, 74, 84, 102, 103, 146, 153, 172, 177, 178).

[17] Jean-Christophe Filliâtre and Andrei Paskevich. 'Why3 — Where Programs Meet Provers'. In: *Programming Languages and Systems*. Ed. by Matthias Felleisen and Philippa Gardner. Berlin, Heidelberg: Springer, 2013, pp. 125–128. DOI: 10.1007/978-3-642-37036-6_8 (cited on pages 1, 8, 17, 172).

[18] A. M. Turing. 'On Computable Numbers, with an Application to the Entscheidungsproblem'. In: *Proceedings of the London Mathematical Society* s2-42.1 (1937), pp. 230–265. DOI: 10.1112/plms/s2-42.1.230 (cited on page 1).

[19] H. G. Rice. 'Classes of Recursively Enumerable Sets and Their Decision Problems'. In: *Trans. Amer. Math. Soc.* 74.2 (1953), pp. 358–366. DOI: 10.1090/S0002-9947-1953-0053041-6 (cited on page 1).

[20] Tahina Ramananandro, Antoine Delignat-Lavaud, Cédric Fournet, Nikhil Swamy, Tej Chajed, Nadim Kobeissi, and Jonathan Protzenko. 'EverParse: Verified Secure Zero-Copy Parsers for Authenticated Message Formats'. In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 1465–1482 (cited on page 1).

[21] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 'HACL*: A Verified Modern Cryptographic Library'. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 1789–1806. DOI: 10.1145/3133956.3134043 (cited on page 1).

[22] Barry Bond, Chris Hawblitzel, Manos Kapritsos, K. Rustan M. Leino, Jacob R. Lorch, Bryan Parno, Ashay Rane, Srinath Setty, and Laure Thompson. 'Vale: Verifying {High-Performance} Cryptographic Assembly Code'. In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, pp. 917–934 (cited on page 1).

[23] Jonathan Protzenko, Bryan Parno, Aymeric Fromherz, Chris Hawblitzel, Marina Polubelova, Karthikeyan Bhargavan, Benjamin Beurdouche, Joonwon Choi, Antoine Delignat-Lavaud, Cédric Fournet, Natalia Kulatova, Tahina Ramananandro, Aseem Rastogi, Nikhil Swamy, Christoph M. Wintersteiger, and Santiago Zanella-Beguelin. 'EverCrypt: A Fast, Verified, Cross-Platform Cryptographic Provider'. In: *2020 IEEE Symposium on Security and Privacy (SP)*. May 2020, pp. 983–1002. DOI: 10.1109/SP40000.2020.00114 (cited on page 1).

[24] Felix A. Wolf, Linard Arquint, Martin Clochard, Wytse Oortwijn, João C. Pereira, and Peter Müller. 'Gobra: Modular Specification and Verification of Go Programs'. In: *Computer Aided Verification*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, 2021, pp. 367–379. DOI: 10.1007/978-3-030-81685-8_17 (cited on pages 1, 4, 5, 10, 178).

[25] João C. Pereira, Tobias Klenze, Sofia Giampietro, Markus Limbeck, Dionysios Spiliopoulos, Felix A. Wolf, Marco Eilers, Christoph Sprenger, David Basin, Peter Müller, and Adrian Perrig. *Protocols to Code: Formal Verification of a Next-Generation Internet Router*. May 2024. DOI: 10.48550/arXiv.2405.06074 (cited on page 2).

[26] Aleks Chakarov, Jaco Geldenhuys, Matthew Heck, Michael Hicks, Sam Huang, Georges-Axel Jaloyan, Anjali Joshi, K. Rustan M. Leino, Mikael Mayer, Sean McLaughlin, Akhilesh Mritunjai, Clement Pit-Claudel, Sorawee Porncharoenwase, Florian Rabe, Marianna Rapoport, Giles Reger, Cody Roux, Neha Rungta, Robin Salkeld, Matthias Schlaipfer, Daniel Schoepe, Johanna Schwartzentruber, Serdar Tasiran, Aaron Tomb, Emina Torlak, Jean-Baptiste Tristan, Lucas Wagner, Michael W. Whalen, Remy Willems, Tongtong Xiang, Tae Joon Byun, Joshua Cohen, Ruijie Fang, Junyoung Jang, Jakob Rath, Hira

Taqdees Syeda, Dominik Wagner, and Yongwei Yuan. 'Formally Verified Cloud-Scale Authorization'. In: *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, Apr. 2025, pp. 2508–2521. DOI: 10.1109/ICSE55347.2025.00166 (cited on page 2).

[27] Michael R. Clarkson and Fred B. Schneider. 'Hyperproperties'. In: *2008 21st IEEE Computer Security Foundations Symposium*. June 2008, pp. 51–65. DOI: 10.1109/CSF.2008.7 (cited on pages 2, 6, 107, 110, 112, 116, 143, 179).

[28] D. McCullough. 'Noninterference and the Composability of Security Properties'. In: *Proceedings. 1988 IEEE Symposium on Security and Privacy*. Apr. 1988, pp. 177–186. DOI: 10.1109/SECPRI.1988.8110 (cited on pages 2, 6, 8, 107).

[29] J. A. Goguen and J. Meseguer. 'Security Policies and Security Models'. In: *1982 IEEE Symposium on Security and Privacy*. Apr. 1982, pp. 11–11. DOI: 10.1109/SP.1982.10014 (cited on pages 2, 6, 110).

[30] José Fragoso Santos, Petar Maksimović, Sacha-Élie Ayoun, and Philippa Gardner. 'Gillian, Part i: A Multi-Language Platform for Symbolic Execution'. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. New York, NY, USA: Association for Computing Machinery, June 2020, pp. 927–942. DOI: 10.1145/3385412.3386014 (cited on pages 2, 4, 5, 9, 17, 21, 36, 47, 86, 103).

[31] Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. 'Iris from the Ground up: A Modular Foundation for Higher-Order Concurrent Separation Logic'. In: *J. Funct. Prog.* 28 (2018), e20. DOI: 10.1017/S0956796818000151 (cited on pages 2, 3, 5, 49, 82, 92, 104, 176).

[32] The Coq Development Team. *The Coq Reference Manual – Release 8.19.0*. 2024 (cited on pages 3, 104).

[33] 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 3, 12, 21, 57, 108, 160, 166).

[34] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. 'The Lean Theorem Prover (System Description)'. In: *Automated Deduction - CADE-25*. Ed. by Amy P. Felty and Aart Middeldorp. Cham: Springer International Publishing, 2015, pp. 378–388. DOI: 10.1007/978-3-319-21401-6_26 (cited on page 3).

[35] 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 '11. New York, NY, USA: Association for Computing Machinery, June 2011, pp. 234–245. DOI: 10.1145/1993498.1993526 (cited on pages 3, 17).

[36] 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 Reasoning* 61.1 (June 2018), pp. 367–422. DOI: 10.1007/s10817-018-9457-5 (cited on pages 3, 17).

[37] 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: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. New York, NY, USA: Association for Computing Machinery, June 2021, pp. 158–174. DOI: 10.1145/3453483.3454036 (cited on pages 3, 17, 49).

[38] Ike Mulder, Robbert Krebbers, and Herman Geuvers. 'Diaframe: Automated Verification of Fine-Grained Concurrent Programs in Iris'. In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. New York, NY, USA: Association for Computing Machinery, June 2022, pp. 809–824. DOI: 10.1145/3519939.3523432 (cited on page 3).

[39] 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 (June 2024), 192:1115–192:1139. DOI: 10.1145/3656422 (cited on page 3).

[40] Litao Zhou, Jianxing Qin, Qinshi Wang, Andrew W. Appel, and Qinxiang Cao. 'VST-A: A Foundationally Sound Annotation Verifier'. In: *Proc. ACM Program. Lang.* 8.POPL (Jan. 2024), 69:2069–69:2098. DOI: 10.1145/3632911 (cited on page 3).

[41] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 'Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning'. In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. New York, NY, USA: Association for Computing Machinery, Jan. 2015, pp. 637–650. DOI: 10.1145/2676726.2676980 (cited on pages 3, 31, 49, 176).

[42] Andrew W. Appel. 'Verified Software Toolchain'. In: *Programming Languages and Systems*. Ed. by Gilles Barthe. Berlin, Heidelberg: Springer, 2011, pp. 1–17. DOI: 10.1007/978-3-642-19718-5_1 (cited on page 3).

[43] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 'Smallfoot: Modular Automatic Assertion Checking with Separation Logic'. In: *Formal Methods for Components and Objects*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever. Berlin, Heidelberg: Springer, 2005, pp. 115–137. DOI: 10.1007/11804192_6 (cited on pages 3, 37).

[44] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 'Symbolic Execution with Separation Logic'. In: *Programming Languages and Systems*. Ed. by Kwangkeun Yi. Berlin, Heidelberg: Springer, 2005, pp. 52–68. DOI: 10.1007/11575467_5 (cited on pages 3, 36).

[45] Thomas Tuerk. 'A Formalisation of Smallfoot in HOL'. In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Berlin, Heidelberg: Springer, 2009, pp. 469–484. DOI: 10.1007/978-3-642-03359-9_32 (cited on page 3).

[46] Andrew W. Appel. 'VeriSmall: Verified Smallfoot Shape Analysis'. In: *Certified Programs and Proofs*. Ed. by Jean-Pierre Jouannaud and Zhong Shao. Berlin, Heidelberg: Springer, 2011, pp. 231–246. DOI: 10.1007/978-3-642-25379-9_18 (cited on page 3).

[47] 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 (Aug. 2022), 97:194–97:224. DOI: 10.1145/3547628 (cited on pages 3, 49).

[48] 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 (Aug. 2021), 85:1–85:30. DOI: 10.1145/3473590 (cited on pages 3, 49).

[49] 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 (Aug. 2020), 121:1–121:30. DOI: 10.1145/3409003 (cited on pages 3, 49).

[50] Gabriel Ebner, Guido Martínez, Aseem Rastogi, Thibault Dardinier, Megan Frisella, Tahina Ramananandro, and Nikhil Swamy. 'PulseCore: An Impredicative Concurrent Separation Logic for Dependently Typed Programs'. In: *Proc. ACM Program. Lang.* 9.PLDI (June 2025), 208:1516–208:1539. DOI: 10.1145/3729311 (cited on pages 3, 14, 104).

[51] Danel Ahman, Cédric Fournet, Cătălin Hriţcu, Kenji Maillard, Aseem Rastogi, and Nikhil Swamy. 'Recalling a Witness: Foundations and Applications of Monotonic State'. In: *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 65:1–65:30. DOI: 10.1145/3158153 (cited on page 3).

[52] Leonardo de Moura and Nikolaj Bjørner. 'Z3: An Efficient SMT Solver'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer, 2008, pp. 337–340. DOI: 10.1007/978-3-540-78800-3_24 (cited on pages 3, 4, 172).

[53] Dino Distefano and Matthew J. Parkinson J. 'jStar: Towards Practical Verification for Java'. In: *SIGPLAN Not.* 43.10 (Oct. 2008), pp. 213–226. DOI: 10.1145/1449955.1449782 (cited on page 4).

[54] 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*. Ed. by Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel. Berlin, Heidelberg: Springer, 2009, pp. 23–42. DOI: 10.1007/978-3-642-03359-9_2 (cited on page 4).

[55] Ruzica Piskac, Thomas Wies, and Damien Zufferey. 'GRASShopper: Complete Heap Verification with Mixed Specifications'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Erika Ábrahám, and Klaus Havelund. Vol. 8413. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 124–139. DOI: 10.1007/978-3-642-54862-8_9 (cited on pages 4, 5).

[56] Thomas Dinsdale-Young, Pedro Da Rocha Pinto, Kristoffer Just Andersen, and Lars Birkedal. 'Caper: Automatic Verification for Fine-Grained Concurrency'. In: *Programming Languages and Systems*. Ed. by Hongseok Yang. Vol. 10201. Berlin, Heidelberg: Springer Berlin Heidelberg, 2017, pp. 420–447. DOI: 10.1007/978-3-662-54434-1_16 (cited on page 4).

[57] Stefan Blom, Saeed Darabi, Marieke Huisman, and Wytse Oortwijn. 'The VerCors Tool Set: Verification of Parallel and Concurrent Software'. In: *Integrated Formal Methods*. Ed. by Nadia Polikarpova and Steve Schneider. Cham: Springer International Publishing, 2017, pp. 102–110. DOI: 10.1007/978-3-319-66845-1_7 (cited on pages 4, 5, 10, 17, 52, 74, 84, 87, 103, 178).

[58] Marco Eilers and Peter Müller. 'Nagini: A Static Verifier for Python'. In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 596–603. DOI: 10.1007/978-3-319-96145-3_33 (cited on pages 4, 5, 10, 178).

[59] Vytautas Astrauskas, Peter Müller, Federico Poli, and Alexander J. Summers. 'Leveraging Rust Types for Modular Specification and Verification'. In: *Proc. ACM Program. Lang.* 3.OOPSLA (Oct. 2019), 147:1–147:30. DOI: 10.1145/3360573 (cited on pages 4, 5, 17, 52, 84, 178, 179).

[60] Christopher Pulte, Dhruv C. Makwana, Thomas Sewell, Kayvan Memarian, Peter Sewell, and Neel Krishnaswami. 'CN: Verifying Systems C Code with Separation-Logic Refinement Types'. In: *Proc. ACM Program. Lang.* 7.POPL (Jan. 2023), 1:1–1:32. DOI: 10.1145/3571194 (cited on page 4).

[61] Can Cebeci, Yonghao Zou, Diyu Zhou, George Candea, and Clément Pit-Claudel. 'Practical Verification of System-Software Components Written in Standard C'. In: *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*. SOSP '24. New York, NY, USA: Association for Computing Machinery, Nov. 2024, pp. 455–472. DOI: 10.1145/3694715.3695980 (cited on page 4).

[62] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. 'Cvc5: A Versatile and Industrial-Strength SMT Solver'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Dana Fisman and Grigore Rosu. Cham: Springer International Publishing, 2022, pp. 415–442. DOI: 10.1007/978-3-030-99524-9_24 (cited on page 4).

[63] Petar Maksimović, Sacha-Élie Ayoun, José Fragoso Santos, and Philippa Gardner. 'Gillian, Part II: Real-World Verification for JavaScript and C'. In: *Computer Aided Verification*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, 2021, pp. 827–850. DOI: 10.1007/978-3-030-81688-9_38 (cited on pages 4, 17, 19, 21, 47).

[64] Sacha-Élie Ayoun, Xavier Denis, Petar Maksimović, and Philippa Gardner. 'A Hybrid Approach to Semi-automated Rust Verification'. In: *Proc. ACM Program. Lang.* 9.PLDI (June 2025), 186:970–186:992. DOI: 10.1145/3729289 (cited on pages 4, 17, 19, 84, 104).

[65] Niels Mommen and Bart Jacobs. *Verification of C++ Programs with VeriFast*. Dec. 2022. DOI: 10.48550/arXiv.2212.13754 (cited on page 4).

[66] Nima Rahimi Foroushaani and Bart Jacobs. *Modular Formal Verification of Rust Programs with Unsafe Blocks*. Dec. 2022. DOI: 10.48550/arXiv.2212.12976 (cited on page 4).

[67] Bart Jacobs, Frédéric Vogels, and Frank Piessens. 'Featherweight VeriFast'. In: *Logical Methods in Computer Science* Volume 11, Issue 3 (Sept. 2015). DOI: 10.2168/LMCS-11(3:19)2015 (cited on pages 4, 19, 48, 49, 176).

[68] Petar Maksimović, José Fragoso Santos, Sacha-Élie Ayoun, and Philippa Gardner. *Gillian: A Multi-Language Platform for Unified Symbolic Analysis*. July 2021. DOI: 10.48550/arXiv.2105.14769 (cited on pages 4, 19, 47).

[69] Andreas Lööw, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Caroline Cronjäger, Petar Maksimović, and Philippa Gardner. 'Compositional Symbolic Execution for Correctness and Incorrectness Reasoning'. In: *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Ed. by Jonathan Aldrich and Guido Salvaneschi. Vol. 313. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 25:1–25:28. DOI: 10.4230/LIPIcs.ECOOP.2024.25 (cited on pages 4, 47).

[70] Jan Smans, Bart Jacobs, and Frank Piessens. 'Implicit Dynamic Frames'. In: *ACM Trans. Program. Lang. Syst.* 34.1 (May 2012), 2:1–2:58. DOI: 10.1145/2160910.2160911 (cited on pages 4, 5, 9, 17, 49, 60, 176).

[71] Conrad Zimmerman, Jenna DiVincenzo, and Jonathan Aldrich. 'Sound Gradual Verification with Symbolic Execution'. In: *Proc. ACM Program. Lang.* 8.POPL (Jan. 2024), 85:2547–85:2576. DOI: 10.1145/3632927 (cited on pages 4, 19, 37, 46–48).

[72] Johannes Bader, Jonathan Aldrich, and Éric Tanter. 'Gradual Program Verification'. In: *Verification, Model Checking, and Abstract Interpretation*. Ed. by Isil Dillig and Jens Palsberg. Cham: Springer International Publishing, 2018, pp. 25–46. DOI: 10.1007/978-3-319-73721-8_2 (cited on page 4).

[73] Thibault Dardinier, Gaurav Parthasarathy, and Peter Müller. 'Verification-Preserving Inlining in Automatic Separation Logic Verifiers'. In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (Apr. 2023), 102:789–102:818. DOI: 10.1145/3586054 (cited on pages 4, 13, 47, 178).

[74] 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 (June 2024), 208:1510–208:1534. DOI: 10.1145/3656438 (cited on pages 4, 5, 10, 13, 14, 19, 20, 22, 26, 36, 39, 40, 46, 48, 102, 178).

[75] Gaurav Parthasarathy, Peter Müller, and Alexander J. Summers. 'Formally Validating a Practical Verification Condition Generator'. In: *Computer Aided Verification*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, 2021, pp. 704–727. DOI: 10.1007/978-3-030-81688-9_33 (cited on pages 5, 17, 48).

[76] Joshua M. Cohen and Philip Johnson-Freyd. 'A Formalization of Core Why3 in Coq'. In: *Proc. ACM Program. Lang.* 8.POPL (Jan. 2024), 60:1789–60:1818. DOI: 10.1145/3632902 (cited on pages 5, 17, 48).

[77] Paolo Herms. 'Certification of a Tool Chain for Deductive Program Verification'. PhD thesis. Université Paris Sud - Paris XI, Jan. 2013 (cited on pages 5, 17, 48).

[78] Stefan Blom and Marieke Huisman. 'Witnessing the Elimination of Magic Wands'. In: *Int J Softw Tools Technol Transfer* 17.6 (Nov. 2015), pp. 757–781. DOI: 10.1007/s10009-015-0372-3 (cited on pages 5, 75, 84).

[79] Malte Schwerhoff and Alexander J. Summers. 'Lightweight Support for Magic Wands in an Automatic Verifier'. In: *29th European Conference on Object-Oriented Programming (ECOOP 2015)*. Ed. by John Tang Boyland. Vol. 37. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015, pp. 614–638. DOI: 10.4230/LIPIcs.ECOOP.2015.614 (cited on pages 5, 22, 75, 84, 87, 89).

[80] 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. Berlin, Heidelberg: Springer, 2009, pp. 195–222. DOI: 10.1007/978-3-642-03829-7_7 (cited on page 5).

[81] John Boyland. 'Checking Interference with Fractional Permissions'. In: *Static Analysis*. Ed. by Radhia Cousot. Berlin, Heidelberg: Springer, 2003, pp. 55–72. DOI: 10.1007/3-540-44898-5_4 (cited on pages 5, 22, 24, 49, 52, 81).

[82] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. 'Permission Accounting in Separation Logic'. In: *SIGPLAN Not.* 40.1 (Jan. 2005), pp. 259–270. DOI: 10.1145/1047659.1040327 (cited on pages 5, 52, 59, 81).

[83] Xuan-Bach Le and Aquinas Hobor. 'Logical Reasoning for Disjoint Permissions'. In: *Programming Languages and Systems*. Ed. by Amal Ahmed. Cham: Springer International Publishing, 2018, pp. 385–414. DOI: 10.1007/978-3-319-89884-1_14 (cited on pages 5, 10, 53–56, 63, 67, 79, 81, 82, 176).

[84] James Brotherston, Diana Costa, Aquinas Hobor, and John Wickerson. 'Reasoning over Permissions Regions in Concurrent Separation Logic'. In: *Computer Aided Verification*. Ed. by Shuvendu K. Lahiri and Chao Wang. Cham: Springer International Publishing, 2020, pp. 203–224. DOI: 10.1007/978-3-030-53291-8_13 (cited on pages 5, 10, 53, 56, 58, 65, 66, 78–81, 176).

[85] Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 'Local Action and Abstract Separation Logic'. In: *22nd Annual IEEE Symposium on Logic in Computer Science (LICS 2007)*. July 2007, pp. 366–378. DOI: 10.1109/LICS.2007.30 (cited on pages 5, 9, 22, 30, 49, 60).

[86] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. 'A Fresh Look at Separation Algebras and Share Accounting'. In: *Programming Languages and Systems*. Ed. by Zhenjiang Hu. Berlin, Heidelberg: Springer, 2009, pp. 161–177. DOI: 10.1007/978-3-642-10672-9_13 (cited on pages 5, 9, 22, 30, 31, 49, 55, 60, 81).

[87] Matthew J. Parkinson and Alexander J. Summers. 'The Relationship Between Separation Logic and Implicit Dynamic Frames'. In: *Logical Methods in Computer Science* Volume 8, Issue 3 (July 2012). DOI: 10.2168/LMCS-8(3:1)2012 (cited on pages 5, 22, 49, 102).

[88] Marcelo Sousa and Isil Dillig. 'Cartesian Hoare Logic for Verifying K-Safety Properties'. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. New York, NY, USA: Association for Computing Machinery, June 2016, pp. 57–69. DOI: 10.1145/2908080.2908092 (cited on pages 6–8, 111, 119, 142, 145, 171, 173, 181, 188).

[89] Peter W. O'Hearn. 'Incorrectness Logic'. In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019), 10:1–10:32. DOI: 10.1145/3371078 (cited on pages 6, 7, 109, 119, 142, 150, 190).

[90] Zhe Zhou, Ashish Mishra, Benjamin Delaware, and Suresh Jagannathan. 'Covering All the Bases: Type-Based Verification of Test Input Generators'. In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023), 157:1244–157:1267. DOI: 10.1145/3591271 (cited on page 6).

[91] Kevin Clancy and Heather Miller. 'Monotonicity Types for Distributed Dataflow'. In: *Proceedings of the Programming Models and Languages for Distributed Computing*. PMLDC '17. New York, NY, USA: Association for Computing Machinery, June 2017, pp. 1–10. DOI: 10.1145/3166089.3166090 (cited on page 6).

[92] Jeffrey Dean and Sanjay Ghemawat. 'MapReduce: Simplified Data Processing on Large Clusters'. In: *Commun. ACM* 51.1 (Jan. 2008), pp. 107–113. DOI: 10.1145/1327452.1327492 (cited on page 6).

[93] Cynthia Dwork. 'Differential Privacy'. In: *Automata, Languages and Programming*. Ed. by Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener. Berlin, Heidelberg: Springer, 2006, pp. 1–12. DOI: 10.1007/11787006_1 (cited on page 6).

[94] Craig Gentry. 'Fully Homomorphic Encryption Using Ideal Lattices'. In: *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*. STOC '09. New York, NY, USA: Association for Computing Machinery, May 2009, pp. 169–178. DOI: 10.1145/1536414.1536440 (cited on page 6).

[95] S. Zdancewic and A.C. Myers. 'Robust Declassification'. In: *Proceedings. 14th IEEE Computer Security Foundations Workshop, 2001.* June 2001, pp. 15–23. DOI: 10.1109/CSFW.2001.930133 (cited on page 6).

[96] Robert Dickerson, Qianchuan Ye, Michael K. Zhang, and Benjamin Delaware. 'RHLE: Modular Deductive Verification of Relational $$\forall \exists $$Properties'. In: *Programming Languages and Systems*. Ed. by Ilya Sergey. Cham: Springer Nature Switzerland, 2022, pp. 67–87. DOI: 10.1007/978-3-031-21037-2_4 (cited on pages 6–8, 109, 111, 143, 145, 171, 173, 194).

[97] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 'Conflict-Free Replicated Data Types'. In: *Stabilization, Safety, and Security of Distributed Systems*. Springer, Berlin, Heidelberg, 2011, pp. 386–400. DOI: 10.1007/978-3-642-24550-3_29 (cited on page 6).

[98] Sam Blackshear, Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 'RacerD: Compositional Static Race Detection'. In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018), 144:1–144:28. DOI: 10.1145/3276514 (cited on pages 7, 8, 142).

[99] Nikos Gorogiannis, Peter W. O'Hearn, and Ilya Sergey. 'A True Positives Theorem for a Static Race Detector'. In: *Proc. ACM Program. Lang.* 3.POPL (Jan. 2019), 57:1–57:29. DOI: 10.1145/3290370 (cited on pages 7, 8, 142).

[100] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. 'Scaling Static Analyses at Facebook'. In: *Commun. ACM* 62.8 (July 2019), pp. 62–70. DOI: 10.1145/3338112 (cited on pages 7, 8, 142).

[101] Quang Loc Le, Azalea Raad, Jules Villard, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 'Finding Real Bugs in Big Programs with Incorrectness Logic'. In: *Proc. ACM Program. Lang.* 6.OOPSLA1 (Apr. 2022), 81:1–81:27. DOI: 10.1145/3527325 (cited on pages 7, 8, 109, 142).

[102] Noam Zilberstein, Derek Dreyer, and Alexandra Silva. 'Outcome Logic: A Unifying Foundation for Correctness and Incorrectness Reasoning'. In: *Proc. ACM Program. Lang.* 7.OOPSLA1 (Apr. 2023), 93:522–93:550. DOI: 10.1145/3586045 (cited on pages 7, 8, 109, 142, 192).

[103] Nick Benton. 'Simple Relational Correctness Proofs for Static Analyses and Program Transformations'. In: *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '04. New York, NY, USA: Association for Computing Machinery, Jan. 2004, pp. 14–25. DOI: 10.1145/964001.964003 (cited on pages 7, 119, 129, 141, 189).

[104] Kenji Maillard, Cătălin Hriţcu, Exequiel Rivas, and Antoine Van Muylder. 'The next 700 Relational Program Logics'. In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019), 4:1–4:33. DOI: 10.1145/3371072 (cited on pages 7, 8, 143).

[105] Timos Antonopoulos, Eric Koskinen, Ton Chanh Le, Ramana Nagasamudram, David A. Naumann, and Minh Ngo. 'An Algebra of Alignment for Relational Verification'. In: *Proc. ACM Program. Lang.* 7.POPL (Jan. 2023), 20:573–20:603. DOI: 10.1145/3571213 (cited on pages 7, 8, 109, 143).

[106] Edsko de Vries and Vasileios Koutavas. 'Reverse Hoare Logic'. In: *Software Engineering and Formal Methods*. Ed. by Gilles Barthe, Alberto Pardo, and Gerardo Schneider. Berlin, Heidelberg: Springer, 2011, pp. 155–171. DOI: 10.1007/978-3-642-24690-6_12 (cited on pages 7, 142, 190).

[107] Toby Murray. *An Under-Approximate Relational Logic: Heralding Logics of Insecurity, Incorrect Implementation & More*. Mar. 2020. DOI: 10.48550/arXiv.2003.04791 (cited on pages 7, 142, 191).

[108] Nissim Francez. 'Product Properties and Their Direct Verification'. In: *Acta Informatica* 20.4 (Dec. 1983), pp. 329–344. DOI: 10.1007/BF00264278 (cited on page 7).

[109] David A. Naumann. 'Thirty-Seven Years of Relational Hoare Logic: Remarks on Its Principles and History'. In: *Leveraging Applications of Formal Methods, Verification and Validation: Engineering Principles*. Ed. by Tiziana Margaria and Bernhard Steffen. Cham: Springer International Publishing, 2020, pp. 93–116. DOI: 10.1007/978-3-030-61470-6_7 (cited on pages 7, 141).

[110] Hongseok Yang. 'Relational Separation Logic'. In: *Theoretical Computer Science*. Festschrift for John C. Reynolds's 70th Birthday 375.1 (May 2007), pp. 308–334. DOI: 10.1016/j.tcs.2006.12.036 (cited on pages 7, 141).

[111] Alejandro Aguirre, Gilles Barthe, Marco Gaboardi, Deepak Garg, and Pierre-Yves Strub. 'A Relational Logic for Higher-Order Programs'. In: *Proc. ACM Program. Lang.* 1.ICFP (Aug. 2017), 21:1–21:29. DOI: 10.1145/3110265 (cited on pages 7, 141).

[112] Torben Amtoft, Sruthi Bandhakavi, and Anindya Banerjee. 'A Logic for Information Flow in Object-Oriented Programs'. In: *SIGPLAN Not.* 41.1 (Jan. 2006), pp. 91–102. DOI: 10.1145/1111320.1111046 (cited on pages 7, 141).

[113] David Costanzo and Zhong Shao. 'A Separation Logic for Enforcing Declarative Information Flow Control Policies'. In: *Principles of Security and Trust*. Ed. by Martín Abadi and Steve Kremer. Berlin, Heidelberg: Springer, 2014, pp. 179–198. DOI: 10.1007/978-3-642-54792-8_10 (cited on pages 7, 141).

[114] Gidon Ernst and Toby Murray. 'SecCSL: Security Concurrent Separation Logic'. In: *Computer Aided Verification*. Ed. by Isil Dillig and Serdar Tasiran. Vol. 11562. Cham: Springer International Publishing, 2019, pp. 208–230. DOI: 10.1007/978-3-030-25543-5_13 (cited on pages 7, 8, 141, 145, 173).

[115] Marco Eilers, Thibault Dardinier, and Peter Müller. 'CommCSL: Proving Information Flow Security for Concurrent Programs Using Abstract Commutativity'. In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023), 175:1682–175:1707. DOI: 10.1145/3591289 (cited on pages 7, 8, 14, 141, 145, 173).

[116] Emanuele D'Osualdo, Azadeh Farzan, and Derek Dreyer. 'Proving Hypersafety Compositionally'. In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (Oct. 2022), 135:289–135:314. DOI: 10.1145/3563298 (cited on pages 7, 142).

[117] Vladimir Gladshtein, Qiyuan Zhao, Willow Ahrens, Saman Amarasinghe, and Ilya Sergey. 'Mechanised Hypersafety Proofs about Structured Data'. In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024), 173:647–173:670. DOI: 10.1145/3656403 (cited on pages 7, 142).

[118] Azalea Raad, Josh Berdine, Hoang-Hai Dang, Derek Dreyer, Peter O'Hearn, and Jules Villard. 'Local Reasoning About the Presence of Bugs: Incorrectness Separation Logic'. In: *Computer Aided Verification*. Ed. by Shuvendu K. Lahiri and Chao Wang. Cham: Springer International Publishing, 2020, pp. 225–252. DOI: 10.1007/978-3-030-53291-8_14 (cited on pages 7, 142).

[119] Azalea Raad, Josh Berdine, Derek Dreyer, and Peter W. O'Hearn. 'Concurrent Incorrectness Separation Logic'. In: *Proc. ACM Program. Lang.* 6.POPL (Jan. 2022), 34:1–34:29. DOI: 10.1145/3498695 (cited on pages 7, 142).

[120] Petar Maksimović, Caroline Cronjäger, Andreas Lööw, Julian Sutherland, and Philippa Gardner. 'Exact Separation Logic: Towards Bridging the Gap Between Verification and Bug-Finding'. In: *37th European Conference on Object-Oriented Programming (ECOOP 2023)*. Ed. by Karim Ali and Guido Salvaneschi. Vol. 263. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 19:1–19:27. DOI: 10.4230/LIPIcs.ECOOP.2023.19 (cited on pages 8, 109, 142).

[121] Raven Beutner. 'Automated Software Verification of Hyperliveness'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Bernd Finkbeiner and Laura Kovács. Cham: Springer Nature Switzerland, 2024, pp. 196–216. DOI: 10.1007/978-3-031-57249-4_10 (cited on pages 8, 143, 145, 173, 181).

[122] G. Barthe, P.R. D'Argenio, and T. Rezk. 'Secure Information Flow by Self-Composition'. In: *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.* June 2004, pp. 100–114. DOI: 10.1109/CSFW.2004.1310735 (cited on pages 8, 143, 145, 172).

[123] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 'Relational Verification Using Product Programs'. In: *FM 2011: Formal Methods*. Ed. by Michael Butler and Wolfram Schulte. Berlin, Heidelberg: Springer, 2011, pp. 200–214. DOI: 10.1007/978-3-642-21437-0_17 (cited on pages 8, 143, 145, 172, 196).

[124] Marco Eilers, Peter Müller, and Samuel Hitz. 'Modular Product Programs'. In: *ACM Trans. Program. Lang. Syst.* 42.1 (Nov. 2019), 3:1–3:37. DOI: 10.1145/3324783 (cited on pages 8, 143, 145, 172, 178).

[125] Gilles Barthe, Juan Manuel Crespo, and César Kunz. 'Beyond 2-Safety: Asymmetric Product Programs for Relational Program Verification'. In: *Logical Foundations of Computer Science*. Ed. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Sergei Artemov, and Anil Nerode. Vol. 7734. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 29–43. DOI: 10.1007/978-3-642-35722-0_3 (cited on pages 8, 143, 145, 173, 196).

[126] Ramana Nagasamudram, Anindya Banerjee, and David A. Naumann. 'The WhyRel Prototype for Modular Relational Verification of Pointer Programs'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Cham: Springer Nature Switzerland, 2023, pp. 133–151. DOI: 10.1007/978-3-031-30820-8_11 (cited on pages 8, 145, 173).

[127] Hiroshi Unno, Tachio Terauchi, and Eric Koskinen. 'Constraint-Based Relational Verification'. In: *Computer Aided Verification*. Ed. by Alexandra Silva and K. Rustan M. Leino. Cham: Springer International Publishing, 2021, pp. 742–766. DOI: 10.1007/978-3-030-81685-8_35 (cited on pages 8, 144, 145, 171, 173).

[128] Raven Beutner and Bernd Finkbeiner. 'Software Verification of Hyperproperties Beyond K-Safety'. In: *Computer Aided Verification*. Ed. by Sharon Shoham and Yakir Vizel. Cham: Springer International Publishing, 2022, pp. 341–362. DOI: 10.1007/978-3-031-13185-1_17 (cited on pages 8, 144, 145, 171, 173).

[129] Thibault Dardinier, Michael Sammler, Gaurav Parthasarathy, Alexander J. Summers, and Peter Müller. *Formal Foundations for Translational Separation Logic Verifiers – Artifact*. Zenodo. Oct. 2024. DOI: 10.5281/zenodo.13938950 (cited on pages 12, 21).

[130] Thibault Dardinier. 'Unbounded Separation Logic'. In: *Archive of Formal Proofs* (Sept. 2022) (cited on pages 12, 57).

[131] Thibault Dardinier, Peter Müller, and Alexander J. Summers. *Fractional Resources in Unbounded Separation Logic (Artifact)*. July 2022. DOI: 10.5281/zenodo.7072457 (cited on pages 12, 57).

[132] Thibault Dardinier. 'Formalization of a Framework for the Sound Automation of Magic Wands'. In: *Archive of Formal Proofs* (May 2022) (cited on pages 12, 85, 89).

[133] Thibault Dardinier. 'A Restricted Definition of the Magic Wand to Soundly Combine Fractions of a Wand'. In: *Archive of Formal Proofs* (May 2022) (cited on pages 12, 85).

[134] Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J. Summers. *Sound Automation of Magic Wands (Artifact)*. Zenodo. May 2022. DOI: 10.5281/zenodo.6525310 (cited on pages 12, 85).

[135] Thibault Dardinier. 'Formalization of Hyper Hoare Logic: A Logic to (Dis-)Prove Program Hyperproperties'. In: *Archive of Formal Proofs* (Apr. 2023) (cited on pages 12, 108).

[136] Thibault Dardinier and Peter Müller. *Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties (Artifact)*. Zenodo. Mar. 2024. DOI: 10.5281/zenodo.10808236 (cited on pages 12, 108).

[137] Thibault Dardinier, Anqi Li, and Peter Müller. *Hypra: A Deductive Program Verifier for Hyperproperties (Artifact)*. Zenodo. July 2024. DOI: 10.5281/zenodo.12671562 (cited on pages 12, 147).

[138] Thibault Dardinier. *Artifact for the PhD Thesis "Formal Foundations for Automated Deductive Verifiers" (2025)*. Zenodo. Sept. 2025. DOI: 10.5281/zenodo.17120270 (cited on page 12).

[139] 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 (Jan. 2025), pp. 569–599. DOI: 10.1145/3704856 (cited on page 12).

[140] Thibault Dardinier, Peter Müller, and Alexander J. Summers. 'Fractional Resources in Unbounded Separation Logic'. In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (Oct. 2022), 163:1066–163:1092. DOI: 10.1145/3563326 (cited on pages 13, 22).

[141] Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Peter Müller, and Alexander J. Summers. 'Sound Automation of Magic Wands'. In: *Computer Aided Verification*. Ed. by Sharon Shoham and Yakir Vizel. Cham: Springer International Publishing, 2022, pp. 130–151. DOI: 10.1007/978-3-031-13188-2_7 (cited on pages 13, 22, 92, 95).

[142] Thibault Dardinier and Peter Müller. 'Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties'. In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024), 207:1485–207:1509. DOI: 10.1145/3656437 (cited on page 13).

[143] Thibault Dardinier, Anqi Li, and Peter Müller. 'Hypra: A Deductive Program Verifier for Hyper Hoare Logic'. In: *Proc. ACM Program. Lang.* 8.OOPSLA2 (Oct. 2024), 316:1279–316:1308. DOI: 10.1145/3689756 (cited on page 13).

[144] Anqi Li. 'An Automatic Program Verifier for Hyperproperties'. MA thesis. ETH Zurich, 2023 (cited on page 13).

[145] David Basin, Thibault Dardinier, Lukas Heimes, Srđan Krstić, Martin Raszyk, Joshua Schneider, and Dmitriy Traytel. 'A Formally Verified, Optimized Monitor for Metric First-Order Dynamic Logic'. In: *Automated Reasoning*. Ed. by Nicolas Peltier and Viorica Sofronie-Stokkermans. Cham: Springer International Publishing, 2020, pp. 432–453. DOI: 10.1007/978-3-030-51074-9_25 (cited on page 13).

[146] Bernhard Kragl and Shaz Qadeer. 'The Civl Verifier'. In: #*PLACEHOLDER_PARENT_METADATA_VALUE*#. TU Wien Academic Press, Oct. 2021, pp. 143–152. DOI: 10.34727/2021/isbn.978-3-85448-046-4_23 (cited on page 17).

[147] Xavier Denis, Jacques-Henri Jourdan, and Claude Marché. 'Creusot: A Foundry for the Deductive Verification of Rust Programs'. In: *Formal Methods and Software Engineering*. Ed. by Adrian Riesco and Min Zhang. Cham: Springer International Publishing, 2022, pp. 90–105. DOI: 10.1007/978-3-031-17244-1_6 (cited on page 17).

[148] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. 'Frama-C: A Software Analysis Perspective'. In: *Form. Asp. Comput.* 27.3 (May 2015), pp. 573–609. DOI: 10.1007/s00165-014-0326-7 (cited on page 17).

[149] Marco Eilers, Malte Schwerhoff, and Peter Müller. *Verification Algorithms for Automated Separation Logic Verifiers*. May 2024. DOI: 10.48550/arXiv.2405.10661 (cited on page 18).

[150] Frank S. de Boer and Marcello Bonsangue. 'Symbolic Execution Formally Explained'. In: *Form. Asp. Comput.* 33.4-5 (Aug. 2021), pp. 617–636. DOI: 10.1007/s00165-020-00527-y (cited on page 20).

[151] Peter Müller, Malte Schwerhoff, and Alexander J. Summers. 'Automatic Verification of Iterated Separating Conjunctions Using Symbolic Execution'. In: *Computer Aided Verification*. Ed. by Swarat Chaudhuri and Azadeh Farzan. Cham: Springer International Publishing, 2016, pp. 405–425. DOI: 10.1007/978-3-319-41528-4_22 (cited on pages 22, 178).

[152] K. Rustan M. Leino and Peter Müller. 'A Basis for Verifying Multi-threaded Programs'. In: *Programming Languages and Systems*. Ed. by Giuseppe Castagna. Berlin, Heidelberg: Springer, 2009, pp. 378–393. DOI: 10.1007/978-3-642-00590-9_27 (cited on pages 24, 52, 61, 176).

[153] Ingrid Rewitzky. 'Binary Multirelations'. In: *Theory and Applications of Relational Structures as Knowledge Instruments: COST Action 274, TARSKI. Revised Papers*. Ed. by Harrie de Swart, Ewa Orłowska, Gunther Schmidt, and Marc Roubens. Berlin, Heidelberg: Springer, 2003, pp. 256–271. DOI: 10.1007/978-3-540-24615-2_12 (cited on page 26).

[154] Armaël Guéneau, Johannes Hostert, Simon Spies, Michael Sammler, Lars Birkedal, and Derek Dreyer. 'Melocoton: A Program Logic for Verified Interoperability Between OCaml and C'. In: *Proc. ACM Program. Lang.* 7.OOPSLA2 (Oct. 2023), 247:716–247:744. DOI: 10.1145/3622823 (cited on pages 26, 48, 176).

[155] Malte H. Schwerhoff. 'Advancing Automated, Permission-Based Program Verification Using Symbolic Execution'. Doctoral Thesis. ETH Zurich, 2016. DOI: 10.3929/ethz-a-010835519 (cited on pages 26, 36–38).

[156] Thibault Dardinier, Michael Sammler, Gaurav Parthasarathy, Alexander J. Summers, and Peter Müller. 'Formal Foundations for Translational Separation Logic Verifiers (Extended Version)'. In: *Proc. ACM Program. Lang.* 9.POPL (Jan. 2025), pp. 569–599. DOI: 10.1145/3704856 (cited on pages 37, 38).

[157] Viktor Vafeiadis. 'Concurrent Separation Logic and Operational Semantics'. In: *Electronic Notes in Theoretical Computer Science*. Twenty-Seventh Conference on the Mathematical Foundations of Programming Semantics (MFPS XXVII) 276 (Sept. 2011), pp. 335–351. DOI: 10.1016/j.entcs.2011.09.029 (cited on pages 42, 183).

[158] Alexander J. Summers and Peter Müller. 'Automating Deductive Verification for Weak-Memory Programs (Extended Version)'. In: *Int J Softw Tools Technol Transfer* 22.6 (Dec. 2020), pp. 709–728. DOI: 10.1007/s10009-020-00559-y (cited on pages 47, 67).

[159] Felix A. Wolf, Malte Schwerhoff, and Peter Müller. 'Concise Outlines for a Complex Logic: A Proof Outline Checker for TaDA'. In: *Form Methods Syst Des* 61.1 (Aug. 2022), pp. 110–136. DOI: 10.1007/s10703-023-00427-w (cited on page 47).

[160] Viktor Vafeiadis and Chinmay Narayan. 'Relaxed Separation Logic: A Program Logic for C11 Concurrency'. In: *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*. OOPSLA '13. New York, NY, USA: Association for Computing Machinery, Oct. 2013, pp. 867–884. DOI: 10.1145/2509136.2509532 (cited on page 47).

[161] Pedro da Rocha Pinto, Thomas Dinsdale-Young, and Philippa Gardner. 'TaDA: A Logic for Time and Data Abstraction'. In: *ECOOP 2014 – Object-Oriented Programming*. Ed. by Richard Jones. Berlin, Heidelberg: Springer, 2014, pp. 207–231. DOI: 10.1007/978-3-662-44202-9_9 (cited on page 47).

[162] Yasunari Watanabe, Kiran Gopinathan, George Pîrlea, Nadia Polikarpova, and Ilya Sergey. 'Certifying the Synthesis of Heap-Manipulating Programs'. In: *Proc. ACM Program. Lang.* 5.ICFP (Aug. 2021), 84:1–84:29. DOI: 10.1145/3473589 (cited on page 47).

[163] Frédéric Vogels, Bart Jacobs, and Frank Piessens. 'A Machine Checked Soundness Proof for an Intermediate Verification Language'. In: *SOFSEM 2009: Theory and Practice of Computer Science*. Ed. by Mogens Nielsen, Antonín Kučera, Peter Bro Miltersen, Catuscia Palamidessi, Petr Tůma, and Frank Valencia. Berlin, Heidelberg: Springer, 2009, pp. 570–581. DOI: 10.1007/978-3-540-95891-8_51 (cited on page 48).

[164] Michael Backes, Cătălin Hriţcu, and Thorsten Tarrach. 'Automatically Verifying Typing Constraints for a Data Processing Language'. In: *Certified Programs and Proofs*. Ed. by Jean-Pierre Jouannaud and Zhong Shao. Berlin, Heidelberg: Springer, 2011, pp. 296–313. DOI: 10.1007/978-3-642-25379-9_22 (cited on page 48).

[165] Jean Fortin. 'BSP-Why, Un Outil Pour La Vérification Déductive de Programmes BSP : Machine-Checked Semantics and Application to Distributed State-Space Algorithms'. PhD thesis. Université Paris-Est, Oct. 2013 (cited on page 48).

[166] Frédéric Vogels, Bart Jacobs, and Frank Piessens. 'A Machine-Checked Soundness Proof for an Efficient Verification Condition Generator'. In: *Proceedings of the 2010 ACM Symposium on Applied Computing*. SAC '10. New York, NY, USA: Association for Computing Machinery, Mar. 2010, pp. 2517–2522. DOI: 10.1145/1774088.1774610 (cited on page 48).

[167] Quentin Garchery. 'A Framework for Proof-carrying Logical Transformations'. In: *Electron. Proc. Theor. Comput. Sci.* 336 (July 2021), pp. 5–23. DOI: 10.4204/EPTCS.336.2 (cited on page 48).

[168] Robert W. Floyd. 'Nondeterministic Algorithms'. In: *J. ACM* 14.4 (Oct. 1967), pp. 636–644. DOI: 10.1145/321420.321422 (cited on page 48).

[169] Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. 'Programming with Angelic Nondeterminism'. In: *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '10. New York, NY, USA: Association for Computing Machinery, Jan. 2010, pp. 339–352. DOI: 10.1145/1706299.1706339 (cited on page 48).

[170] Michael Sammler, Simon Spies, Youngju Song, Emanuele D'Osualdo, Robbert Krebbers, Deepak Garg, and Derek Dreyer. 'DimSum: A Decentralized Approach to Multi-language Semantics and Verification'. In: *Proc. ACM Program. Lang.* 7.POPL (Jan. 2023), 27:775–27:805. DOI: 10.1145/3571220 (cited on pages 48, 176).

[171] Youngju Song, Minki Cho, Dongjae Lee, Chung-Kil Hur, Michael Sammler, and Derek Dreyer. 'Conditional Contextual Refinement'. In: *Proc. ACM Program. Lang.* 7.POPL (Jan. 2023), 39:1121–39:1151. DOI: 10.1145/3571232 (cited on pages 48, 176).

[172] Simon Spies, Niklas Mück, Haoyi Zeng, Michael Sammler, Andrea Lattuada, Peter Müller, and Derek Dreyer. 'Destabilizing Iris'. In: *Proc. ACM Program. Lang.* 9.PLDI (June 2025), 181:848–181:873. DOI: 10.1145/3729284 (cited on pages 49, 176).

[173] Toshiyuki Maeda, Haruki Sato, and Akinori Yonezawa. 'Extended Alias Type System Using Separating Implication'. In: *Proceedings of the 7th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. TLDI '11. New York, NY, USA: Association for Computing Machinery, Jan. 2011, pp. 29–42. DOI: 10.1145/1929553.1929559 (cited on pages 52, 83, 84).

[174] Thomas Tuerk. 'Local Reasoning about While-Loops'. In: *VSTTE* 29 (2010), p. 2010 (cited on pages 52, 83, 84, 179).

[175] Neelakantan R. Krishnaswami. 'Reasoning about Iterators with Separation Logic'. In: *Proceedings of the 2006 Conference on Specification and Verification of Component-Based Systems*. SAVCBS '06. New York, NY, USA: Association for Computing Machinery, Nov. 2006, pp. 83–86. DOI: 10.1145/1181195.1181213 (cited on pages 52, 84).

[176] Christian Haack and Clément Hurlin. 'Resource Usage Protocols for Iterators.' In: *JOT* 8.4 (2009), p. 55. DOI: 10.5381/jot.2009.8.4.a3 (cited on pages 52, 84).

[177] Jonas Braband Jensen, Lars Birkedal, and Peter Sestoft. 'Modular Verification of Linked Lists with Views via Separation Logic.' In: *JOT* 10 (2011), 2:1. DOI: 10.5381/jot.2011.10.1.a2 (cited on pages 52, 84).

[178] Bart Jacobs and Frank Piessens. 'Expressive Modular Fine-Grained Concurrency Specification'. In: *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '11. New York, NY, USA: Association for Computing Machinery, Jan. 2011, pp. 271–282. DOI: 10.1145/1926385.1926417 (cited on page 67).

[179] Willem Penninckx, Bart Jacobs, and Frank Piessens. 'Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs'. In: *Programming Languages and Systems*. Ed. by Jan Vitek. Berlin, Heidelberg: Springer, 2015, pp. 158–182. DOI: 10.1007/978-3-662-46669-8_7 (cited on page 69).

[180] Alfred Tarski. 'A Lattice-Theoretical Fixpoint Theorem and Its Applications.' In: *Pacific Journal of Mathematics* 5.2 (Jan. 1955), pp. 285–309 (cited on page 70).

[181] Patrick Cousot and Radhia Cousot. 'Constructive Versions of Tarski's Fixed Point Theorems'. In: *Pacific J. Math.* 82.1 (May 1979), pp. 43–57. DOI: 10.2140/pjm.1979.82.43 (cited on page 71).

[182] Qinxiang Cao, Shengyi Wang, Aquinas Hobor, and Andrew W. Appel. *Proof Pearl: Magic Wand as Frame*. Sept. 2019. DOI: 10.48550/arXiv.1909.08789 (cited on pages 78, 84).

[183] Jules Villard, Étienne Lozes, and Cristiano Calcagno. 'Proving Copyless Message Passing'. In: *Programming Languages and Systems*. Ed. by Zhenjiang Hu. Berlin, Heidelberg: Springer, 2009, pp. 194–209. DOI: 10.1007/978-3-642-10672-9_15 (cited on page 79).

[184] Christian J. Bell, Andrew W. Appel, and David Walker. 'Concurrent Separation Logic for Pipelined Parallelization'. In: *Static Analysis*. Ed. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Radhia Cousot, and Matthieu Martel. Vol. 6337. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 151–166. DOI: 10.1007/978-3-642-15769-1_10 (cited on page 79).

[185] K. Rustan M. Leino, Peter Müller, and Jan Smans. 'Deadlock-Free Channels and Locks'. In: *Programming Languages and Systems*. Ed. by Andrew D. Gordon. Berlin, Heidelberg: Springer, 2010, pp. 407–426. DOI: 10.1007/978-3-642-11957-6_22 (cited on page 79).

[186] Aquinas Hobor and Cristian Gherghina. 'Barriers in Concurrent Separation Logic: Now With Tool Support!' In: *Logical Methods in Computer Science* Volume 8, Issue 2 (Apr. 2012). DOI: 10.2168/LMCS-8(2:2)2012 (cited on page 79).

[187] Matthew Parkinson. 'Local Reasoning for Java'. PhD thesis. Jan. 2005 (cited on page 81).

[188] John Tang Boyland. 'Semantics of Fractional Permissions with Nesting'. In: *ACM Trans. Program. Lang. Syst.* 32.6 (Aug. 2010), 22:1–22:33. DOI: 10.1145/1749608.1749611 (cited on pages 81, 82).

[189] Bor-Yuh Evan Chang and Xavier Rival. 'Relational Inductive Shape Analysis'. In: *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. New York, NY, USA: Association for Computing Machinery, Jan. 2008, pp. 247–260. DOI: 10.1145/1328438.1328469 (cited on pages 82, 104).

[190] Matthew Parkinson and Gavin Bierman. 'Separation Logic and Abstraction'. In: *SIGPLAN Not.* 40.1 (Jan. 2005), pp. 247–258. DOI: 10.1145/1047659.1040326 (cited on page 82).

[191] Andrew W. Appel and David McAllester. 'An Indexed Model of Recursive Types for Foundational Proof-Carrying Code'. In: *ACM Trans. Program. Lang. Syst.* 23.5 (Sept. 2001), pp. 657–683. DOI: 10.1145/504709.504712 (cited on pages 82, 178).

[192] Amal Ahmed. 'Step-Indexed Syntactic Logical Relations for Recursive and Quantified Types'. In: *Programming Languages and Systems*. Ed. by Peter Sestoft. Berlin, Heidelberg: Springer, 2006, pp. 69–83. DOI: 10.1007/11693024_6 (cited on page 82).

[193] Kasper Svendsen and Lars Birkedal. 'Impredicative Concurrent Abstract Predicates'. In: *Programming Languages and Systems*. Ed. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, and Zhong Shao. Vol. 8410. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 149–168. DOI: 10.1007/978-3-642-54833-8_9 (cited on page 82).

[194] Hongseok Yang. 'An Example of Local Reasoning in BI Pointer Logic: The Schorr-Waite Graph Marking Algorithm'. In: *SPACE*. 2001 (cited on page 84).

[195] Mike Dodds, Suresh Jagannathan, and Matthew J. Parkinson. 'Modular Reasoning for Deterministic Parallelism'. In: *SIGPLAN Not.* 46.1 (Jan. 2011), pp. 259–270. DOI: 10.1145/1925844.1926416 (cited on page 84).

[196] Rémi Brochenin, Stéphane Demri, and Etienne Lozes. 'On the Almighty Wand'. In: *Information and Computation* 211 (Feb. 2012), pp. 106–137. DOI: 10.1016/j.ic.2011.12.003 (cited on pages 84, 104).

[197] Andreas Lööw, Daniele Nantes-Sobrinho, Sacha-Élie Ayoun, Petar Maksimović, and Philippa Gardner. 'Matching Plans for Frame Inference in Compositional Reasoning'. In: *38th European Conference on Object-Oriented Programming (ECOOP 2024)*. Ed. by Jonathan Aldrich and Guido Salvaneschi. Vol. 313. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024, 26:1–26:20. DOI: 10.4230/LIPIcs.ECOOP.2024.26 (cited on pages 84, 103).

[198] Thibault Dardinier, Gaurav Parthasarathy, Noé Weeks, Alexanders J. Summers, and Peter Müller. *Sound Automation of Magic Wands (Extended Version)*. Aug. 2022. DOI: 10.48550/arXiv.2205.11325 (cited on pages 86, 90, 185).

[199] Marieke Huisman, Vladimir Klebanov, and Rosemary Monahan. 'VerifyThis 2012: A Program Verification Competition'. In: *Int J Softw Tools Technol Transfer* 17.6 (Nov. 2015), pp. 647–657. DOI: 10.1007/s10009-015-0396-8 (cited on page 86).

[200] Wonyeol Lee and Sungwoo Park. 'A Proof System for Separation Logic with Magic Wand'. In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. New York, NY, USA: Association for Computing Machinery, Jan. 2014, pp. 477–490. DOI: 10.1145/2535838.2535871 (cited on page 104).

[201] Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. 'Shape Analysis with Structural Invariant Checkers'. In: *Static Analysis*. Ed. by Hanne Riis Nielson and Gilberto Filé. Berlin, Heidelberg: Springer, 2007, pp. 384–401. DOI: 10.1007/978-3-540-74061-2_24 (cited on page 104).

[202] Robbert Krebbers, Jacques-Henri Jourdan, Ralf Jung, Joseph Tassarotti, Jan-Oliver Kaiser, Amin Timany, Arthur Charguéraud, and Derek Dreyer. 'MoSeL: A General, Extensible Modal Framework for Interactive Proofs in Separation Logic'. In: *Proc. ACM Program. Lang.* 2.ICFP (July 2018), 77:1–77:30. DOI: 10.1145/3236772 (cited on page 104).

[203] C. E. Shannon. 'A Mathematical Theory of Communication'. In: *The Bell System Technical Journal* 27.3 (July 1948), pp. 379–423. DOI: 10.1002/j.1538-7305.1948.tb01338.x (cited on page 107).

[204] Mounir Assaf, David A. Naumann, Julien Signoles, Éric Totel, and Frédéric Tronel. 'Hypercollecting Semantics and Its Application to Static Analysis of Information Flow'. In: *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. POPL '17. New York, NY, USA: Association for Computing Machinery, Jan. 2017, pp. 874–887. DOI: 10.1145/3009837.3009889 (cited on pages 107, 144, 173).

[205] Hirotoshi Yasuoka and Tachio Terauchi. 'On Bounding Problems of Quantitative Information Flow'. In: *Computer Security – ESORICS 2010*. Ed. by Dimitris Gritzalis, Bart Preneel, and Marianthi Theoharidou. Berlin, Heidelberg: Springer, 2010, pp. 357–372. DOI: 10.1007/978-3-642-15497-3_22 (cited on page 107).

[206] Geoffrey Smith. 'On the Foundations of Quantitative Information Flow'. In: *Foundations of Software Science and Computational Structures*. Ed. by Luca De Alfaro. Vol. 5504. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 288–302. DOI: 10.1007/978-3-642-00596-1_21 (cited on page 107).

[207] Thibault Dardinier and Peter Müller. 'Hyper Hoare Logic: (Dis-)Proving Program Hyperproperties (Extended Version)'. In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024), pp. 1485–1509. DOI: 10.1145/3656437 (cited on pages 107, 190, 195, 196).

[208] Dennis Volpano, Cynthia Irvine, and Geoffrey Smith. 'A Sound Type System for Secure Flow Analysis'. In: *J. Comput. Secur.* 4.2-3 (Jan. 1996), pp. 167–187 (cited on pages 110, 141, 149, 173, 180).

[209] Thomas Kleymann. 'Hoare Logic and Auxiliary Variables'. In: *Form. Asp. Comput.* 11.5 (Dec. 1999), pp. 541–566. DOI: 10.1007/s001650050057 (cited on page 110).

[210] Daryl McCullough. 'Specifications for Multi-Level Security and a Hook-Up'. In: *1987 IEEE Symposium on Security and Privacy*. Apr. 1987, pp. 161–161. DOI: 10.1109/SP.1987.10009 (cited on pages 112, 149, 173).

[211] J. McLean. 'A General Theory of Composition for a Class of "Possibilistic" Properties'. In: *IEEE Transactions on Software Engineering* 22.1 (Jan. 1996), pp. 53–67. DOI: 10.1109/32.481534 (cited on pages 112, 173).

[212] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. 'Proving Non-Termination'. In: *SIGPLAN Not.* 43.1 (Jan. 2008), pp. 147–158. DOI: 10.1145/1328897.1328459 (cited on page 118).

[213] Azalea Raad, Julien Vanegue, and Peter O'Hearn. 'Non-Termination Proving at Scale'. In: *Proc. ACM Program. Lang.* 8.OOPSLA2 (Oct. 2024), 280:246–280:274. DOI: 10.1145/3689720 (cited on page 119).

[214] Flavio Ascari, Roberto Bruni, Roberta Gori, and Francesco Logozzo. *Sufficient Incorrectness Logic: SIL and Separation SIL*. Jan. 2024. DOI: 10.48550/arXiv.2310.18156 (cited on pages 119, 142, 192).

[215] N. G de Bruijn. 'Lambda Calculus Notation with Nameless Dummies, a Tool for Automatic Formula Manipulation, with Application to the Church-Rosser Theorem'. In: *Indagationes Mathematicae (Proceedings)* 75.5 (Jan. 1972), pp. 381–392. DOI: 10.1016/1385-7258(72)90034-0 (cited on page 125).

[216] Tachio Terauchi and Alex Aiken. 'Secure Information Flow as a Safety Problem'. In: *Static Analysis*. Ed. by Chris Hankin and Igor Siveroni. Berlin, Heidelberg: Springer, 2005, pp. 352–367. DOI: 10.1007/11547662_24 (cited on pages 129, 143, 172).

[217] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 'A Logic for Locally Complete Abstract Interpretations'. In: *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. June 2021, pp. 1–13. DOI: 10.1109/LICS52264.2021.9470608 (cited on page 142).

[218] Roberto Bruni, Roberto Giacobazzi, Roberta Gori, and Francesco Ranzato. 'A Correctness and Incorrectness Program Logic'. In: *J. ACM* 70.2 (Mar. 2023), 15:1–15:45. DOI: 10.1145/3582267 (cited on page 142).

[219] Bernhard Möller, Peter O'Hearn, and Tony Hoare. 'On Algebra of Program Correctness and Incorrectness'. In: *Relational and Algebraic Methods in Computer Science: 19th International Conference, RAMiCS 2021, Marseille, France, November 2–5, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, Nov. 2021, pp. 325–343. DOI: 10.1007/978-3-030-88701-8_20 (cited on page 142).

[220] David Harel, G. Goos, J. Hartmanis, P. Brinch Hansen, D. Gries, C. Moler, G. Seegmüller, J. Stoer, and N. Wirth, eds. *First-Order Dynamic Logic*. Vol. 68. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 1979 (cited on page 142).

[221] Lyle Harold Ramshaw. *Formalizing the Analysis of Algorithms*. Stanford University, 1979 (cited on page 143).

[222] J. I. den Hartog. 'Verifying Probabilistic Programs Using a Hoare like Logic'. In: *Advances in Computing Science — ASIAN'99*. Ed. by P. S. Thiagarajan and Roland Yap. Berlin, Heidelberg: Springer, 1999, pp. 113–125. DOI: 10.1007/3-540-46674-6_11 (cited on page 143).

[223] Ricardo Corin and Jerry den Hartog. 'A Probabilistic Hoare-style Logic for Game-Based Cryptographic Proofs'. In: *Automata, Languages and Programming*. Ed. by Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener. Berlin, Heidelberg: Springer, 2006, pp. 252–263. DOI: 10.1007/11787006_22 (cited on page 143).

[224] Gilles Barthe, Thomas Espitau, Marco Gaboardi, Benjamin Grégoire, Justin Hsu, and Pierre-Yves Strub. 'An Assertion-Based Program Logic for Probabilistic Programs'. In: *Programming Languages and Systems*. Ed. by Amal Ahmed. Cham: Springer International Publishing, 2018, pp. 117–144. DOI: 10.1007/978-3-319-89884-1_5 (cited on page 143).

[225] Gilles Barthe, Justin Hsu, and Kevin Liao. 'A Probabilistic Separation Logic'. In: *Proc. ACM Program. Lang.* 4.POPL (Dec. 2019), 55:1–55:30. DOI: 10.1145/3371123 (cited on page 143).

[226] Robert Rand and Steve Zdancewic. 'VPHL: A Verified Partial-Correctness Logic for Probabilistic Programs'. In: *Electronic Notes in Theoretical Computer Science*. The 31st Conference on the Mathematical Foundations of Programming Semantics (MFPS XXXI). 319 (Dec. 2015), pp. 351–367. DOI: 10.1016/j.entcs.2015.12.021 (cited on page 143).

[227] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. 'Formal Certification of Code-Based Cryptographic Proofs'. In: *SIGPLAN Not.* 44.1 (Jan. 2009), pp. 90–101. DOI: 10.1145/1594834.1480894 (cited on page 143).

[228] Gilles Barthe, Marco Gaboardi, Emilio Jesus Gallego Arias, Justin Hsu, Cesar Kunz, and Pierre-Yves Strub. 'Proving Differential Privacy in Hoare Logic'. In: *2014 IEEE 27th Computer Security Foundations Symposium (CSF)*. IEEE Computer Society, July 2014, pp. 411–424. DOI: 10.1109/CSF.2014.36 (cited on page 143).

[229] Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. 'Temporal Logics for Hyperproperties'. In: *Principles of Security and Trust*. Ed. by David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor, Oscar Nierstrasz, C. Pandu Rangan, Bernhard Steffen, Madhu Sudan, Demetri Terzopoulos, Doug Tygar, Moshe Y. Vardi, Gerhard Weikum, Martín Abadi, and Steve Kremer. Vol. 8414. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 265–284. DOI: 10.1007/978-3-642-54792-8_15 (cited on pages 144, 173).

[230] Norine Coenen, Bernd Finkbeiner, César Sánchez, and Leander Tentrup. 'Verifying Hyperliveness'. In: *Computer Aided Verification*. Ed. by Isil Dillig and Serdar Tasiran. Cham: Springer International Publishing, 2019, pp. 121–139. DOI: 10.1007/978-3-030-25540-4_7 (cited on pages 144, 173).

[231] Tzu-Han Hsu, César Sánchez, and Borzoo Bonakdarpour. 'Bounded Model Checking for Hyperproperties'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Jan Friso Groote and Kim Guldstrand Larsen. Cham: Springer International Publishing, 2021, pp. 94–112. DOI: 10.1007/978-3-030-72016-2_6 (cited on pages 144, 173).

[232] Raven Beutner and Bernd Finkbeiner. 'AutoHyper: Explicit-State Model Checking for HyperLTL'. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Sriram Sankaranarayanan and Natasha Sharygina. Cham: Springer Nature Switzerland, 2023, pp. 145–163. DOI: 10.1007/978-3-031-30823-9_8 (cited on pages 144, 173).

[233] Gilles Barthe, Renate Eilers, Pamina Georgiou, Bernhard Gleiss, Laura Kovacs, and Matteo Maffei. 'Verifying Relational Properties Using Trace Logic'. In: *2019 Formal Methods in Computer Aided Design (FMCAD)*. IEEE, 2019. DOI: 10.23919/fmcad.2019.8894277 (cited on pages 144, 173).

[234] Patrick Cousot and Radhia Cousot. 'Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints'. In: *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '77. New York, NY, USA: Association for Computing Machinery, Jan. 1977, pp. 238–252. DOI: 10.1145/512950.512973 (cited on pages 144, 173).

[235] Michele Pasqua. 'Hyper Static Analysis of Programs – An Abstract Interpretation-Based Framework for Hyperproperties Verification'. PhD thesis. May 2019. DOI: 10.5281/zenodo.6584085 (cited on page 144).

[236] David A. Naumann and Minh Ngo. 'Whither Specifications as Programs'. In: *Unifying Theories of Programming*. Ed. by Pedro Ribeiro and Augusto Sampaio. Cham: Springer International Publishing, 2019, pp. 39–61. DOI: 10.1007/978-3-030-31038-7_3 (cited on page 144).

[237] David Detlefs, Greg Nelson, and James B. Saxe. 'Simplify: A Theorem Prover for Program Checking'. In: *J. ACM* 52.3 (May 2005), pp. 365–473. DOI: 10.1145/1066100.1066102 (cited on page 155).

[238] K. Rustan M. Leino and Rosemary Monahan. 'Reasoning about Comprehensions with First-Order SMT Solvers'. In: *Proceedings of the 2009 ACM Symposium on Applied Computing*. SAC '09. New York, NY, USA: Association for Computing Machinery, Mar. 2009, pp. 615–622. DOI: 10.1145/1529282.1529411 (cited on page 159).

[239] Paul Winkler. 'Improving a Deductive Program Verifier for Hyperproperties'. Bachelor's Thesis. ETH Zurich, 2025 (cited on page 171).

[240] Martín Abadi and Leslie Lamport. 'The Existence of Refinement Mappings'. In: *Theoretical Computer Science* 82.2 (May 1991), pp. 253–284. DOI: 10.1016/0304-3975(91)90224-P (cited on pages 173, 194, 196).

[241] Azadeh Farzan and Anthony Vandikas. 'Automated Hypersafety Verification'. In: *Computer Aided Verification*. Ed. by Isil Dillig and Serdar Tasiran. Cham: Springer International Publishing, 2019, pp. 200–218. DOI: 10.1007/978-3-030-25540-4_11 (cited on page 173).

[242] Shachar Itzhaky, Sharon Shoham, and Yakir Vizel. 'Hyperproperty Verification as CHC Satisfiability'. In: *Programming Languages and Systems*. Ed. by Stephanie Weirich. Cham: Springer Nature Switzerland, 2024, pp. 212–241. DOI: 10.1007/978-3-031-57267-8_9 (cited on page 173).

[243] Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. 'Horn Clause Solvers for Program Verification'. In: *Fields of Logic and Computation II: Essays Dedicated to Yuri Gurevich on the Occasion of His 75th Birthday*. Ed. by Lev D. Beklemishev, Andreas Blass, Nachum Dershowitz, Bernd Finkbeiner, and Wolfram Schulte. Cham: Springer International Publishing, 2015, pp. 24–51. DOI: 10.1007/978-3-319-23534-9_2 (cited on page 173).

[244] D. Volpano and G. Smith. 'Eliminating Covert Flows with Minimum Typings'. In: *Proceedings 10th Computer Security Foundations Workshop*. June 1997, pp. 156–168. DOI: 10.1109/CSFW.1997.596807 (cited on page 173).

[245] Edmund M. Clarke. 'Model Checking'. In: *Foundations of Software Technology and Theoretical Computer Science*. Ed. by S. Ramesh and G. Sivakumar. Berlin, Heidelberg: Springer, 1997, pp. 54–56. DOI: 10.1007/BFb0058022 (cited on page 173).

[246] Bernd Finkbeiner, Markus N. Rabe, and César Sánchez. 'Algorithms for Model Checking HyperLTL and HyperCTL $*$'. In: *Computer Aided Verification*. Ed. by Daniel Kroening and Corina S. Păsăreanu. Vol. 9206. Cham: Springer International Publishing, 2015, pp. 30–48. DOI: 10.1007/978-3-319-21690-4_3 (cited on page 173).

[247] Georg Cantor. 'Über eine elementare Frage der Mannigfaltigkeitslehre.' In: *Jahresbericht der Deutschen Mathematiker-Vereinigung* 1 (1890/91), pp. 72–78 (cited on page 177).

[248] Ioannis T. Kassios. 'Dynamic Frames: Support for Framing, Dependencies and Sharing Without Restrictions'. In: *FM 2006: Formal Methods*. Ed. by Jayadev Misra, Tobias Nipkow, and Emil Sekerinski. Berlin, Heidelberg: Springer, 2006, pp. 268–283. DOI: 10.1007/11813040_19 (cited on page 177).

[249] Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. 'Probabilistic Programming'. In: *Future of Software Engineering Proceedings*. FOSE 2014. New York, NY, USA: Association for Computing Machinery, May 2014, pp. 167–181. DOI: 10.1145/2593882.2593900 (cited on page 177).

[250] Gaurav Parthasarathy. 'Formally Validating Translational Program Verifiers'. Doctoral Thesis. ETH Zurich, 2024. DOI: 10.3929/ethz-b-000716048 (cited on page 177).

[251] Stefan Heule, Ioannis T. Kassios, Peter Müller, and Alexander J. Summers. 'Verification Condition Generation for Permission Logics with Abstract Predicates and Abstraction Functions'. In: *ECOOP 2013 – Object-Oriented Programming*. Ed. by Giuseppe Castagna. Berlin, Heidelberg: Springer, 2013, pp. 451–476. DOI: 10.1007/978-3-642-39038-8_19 (cited on page 178).

[252] Vytautas Astrauskas, Aurel Bílý, Jonáš Fiala, Zachary Grannan, Christoph Matheja, Peter Müller, Federico Poli, and Alexander J. Summers. 'The Prusti Project: Formal Verification for Rust'. In: *NASA Formal Methods*. Ed. by Jyotirmoy V. Deshmukh, Klaus Havelund, and Ivan Perez. Cham: Springer International Publishing, 2022, pp. 88–108. DOI: 10.1007/978-3-031-06773-0_5 (cited on page 178).

[253] Jérôme Dohrau. 'Automatic Inference of Permission Specifications'. Doctoral Thesis. ETH Zurich, 2022. DOI: 10.3929/ethz-b-000588977 (cited on page 178).

[254] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. 'Software Verification Using K-Induction'. In: *Static Analysis*. Ed. by Eran Yahav. Berlin, Heidelberg: Springer, 2011, pp. 351–368. DOI: 10.1007/978-3-642-23702-7_26 (cited on page 179).

[255] I. D. Scherson and S. Sen. 'Parallel Sorting in Two-Dimensional VLSI Models of Computation'. In: *IEEE Trans. Comput.* 38.2 (Feb. 1989), pp. 238–249. DOI: 10.1109/12.16500 (cited on page 179).

[256] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd Ed.) Sorting and Searching*. USA: Addison Wesley Longman Publishing Co., Inc., 1998 (cited on page 179).

[257] Jean-Christophe Filliâtre, Léon Gondelman, and Andrei Paskevich. 'The Spirit of Ghost Code'. In: *Computer Aided Verification*. Ed. by Armin Biere and Roderick Bloem. Cham: Springer International Publishing, 2014, pp. 1–16. DOI: 10.1007/978-3-319-08867-9_1 (cited on page 179).

[258] James Li, Noam Zilberstein, and Alexandra Silva. *Total Outcome Logic: Unified Reasoning for a Taxonomy of Program Logics*. June 2025. DOI: 10.48550/arXiv.2411.00197 (cited on page 180).

[259] Edsger W. Dijkstra. 'Guarded Commands, Nondeterminacy and Formal Derivation of Programs'. In: *Commun. ACM* 18.8 (Aug. 1975), pp. 453–457. DOI: 10.1145/360933.360975 (cited on page 180).