TU WIEN Informatics

Towards Automating Induction for Software Verification

Guiding Inductive Reasoning in Superposition-based Theorem Proving

DISSERTATION

zur Erlangung des akademischen Grades

Doktorin der Technischen Wissenschaften

eingereicht von

Dipl.-Ing. Pamina Georgiou, BSc
Matrikelnummer 01125496

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Dr.techn. Laura Kovács, MSc
Zweitbetreuung: Univ.Prof. Dipl.-Ing. Georg Weissenbacher, D.Phil.

Diese Dissertation haben begutachtet:

Reiner Hähnle	Martina Seidl

Wien, 4. März 2024

Pamina Georgiou

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Towards Automating Induction for Software Verification

Guiding Inductive Reasoning in Superposition-based Theorem Proving

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktorin der Technischen Wissenschaften

by

Dipl.-Ing. Pamina Georgiou, BSc
Registration Number 01125496

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Dr.techn. Laura Kovács, MSc
Second advisor: Univ.Prof. Dipl.-Ing. Georg Weissenbacher, D.Phil.

The dissertation has been reviewed by:

Reiner Hähnle	Martina Seidl

Vienna, March 4, 2024

Pamina Georgiou

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Dipl.-Ing. Pamina Georgiou, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 4. März 2024

Pamina Georgiou

Danksagung

Mein herzlicher Dank gilt meiner Betreuerin Laura Kovács, deren unerschütterliche Unterstützung, unschätzbare Anleitung und vorbildliche Mentorrolle während dieser Reise von entscheidender Bedeutung waren. Ich bin ebenso meinem Co-Betreuer Georg Weissenbacher, und meinen Co-Autoren Bernhard Gleiss, Michael Rawson, Ahmed Bhayat, Clemens Eisenhofer und Martón Hajdu für ihre Expertise, intellektuellen Austausch und konstruktives Feedback zutiefst verpflichtet. In diesem Sinne möchte ich mich auch bei meinen Gutachtern Reiner Hähnle und Martina Seidl für die ausführlichen Reviews und das umfangreiche Feedback zu dieser Arbeit bedanken.

Ich bin zutiefst dankbar für die Unterstützung und Kameradschaft, die mir von meinen Kollegen Petra, Sarah, Marcel, Sophie, Giovanni, Sanja, Federica, Anja, Michael und Matthias unter vielen anderen an der TU Wien entgegengebracht wurde. Ihre Ermutigung, aufschlussreichen Gespräche und Freundschaft waren eine ständige Bereicherung. Besonderer Dank gilt auch Beatrix für ihre unglaubliche Unterstützung bei allen organisatorischen und bürokratischen Fragen und Anliegen.

Ich schulde meinen Eltern Brigitte und Nikolaos, dabei insbesondere meiner Mutter Brigitte, einen großen Dank, deren grenzenlose Ermutigung, unerschütterlicher Glaube an mein Potenzial und unermüdliche Unterstützung die Eckpfeiler meiner akademischen Reise waren. Ihre Ermutigung, meine Leidenschaften und Träume zu verfolgen, war die treibende Kraft hinter diesem Erfolg.

Darüber hinaus möchte ich meinen Großeltern, Hilde und Günther, für ihre kontinuierliche Unterstützung während meiner gesamten Ausbildung von Herzen danken. Ihre Großzügigkeit und ihr Glaube an meine Fähigkeiten haben die mit dem Studium verbundenen Belastungen gemildert und es mir ermöglicht, mich voll und ganz auf mein Studium zu konzentrieren.

Besonderer Dank gilt meinem Partner Stefan, dessen Liebe, Unterstützung und kontinuierlicher Glaube an meine Fähigkeiten mich durch die Höhen und Tiefen dieses Vorhabens getragen haben, insbesondere in den letzten Zügen der Niederschrift.

An meine lieben Freunde, die mich auf dieser akademischen Reise begleitet haben, insbesondere Andjela, Corinna, Sandra und Sophie, eure Unterstützung und Ermutigung waren eine Quelle der Kraft während der Triumphe und Herausforderungen des Doktorats. Ihre Anwesenheit hat diese Erfahrung noch erfüllender und denkwürdiger gemacht.

Schließlich möchte ich all jenen meinen aufrichtigen Dank aussprechen, deren Namen hier vielleicht nicht genannt werden, die aber durch ihre Unterstützung einen wesentlichen Beitrag zur Entstehung dieser Arbeit geleistet haben.

Diese Arbeit wäre ohne die kollektive Unterstützung aller oben genannten Personen nicht möglich gewesen. Ich danke euch, dass ihr ein wesentlicher Bestandteil dieser transformativen Reise waren.

Acknowledgements

I extend my heartfelt gratitude to my main supervisor, Laura Kovács, whose unwavering support, invaluable guidance, and exemplary mentorship have been instrumental throughout this journey. I am equally indebted to my co-supervisor, Georg Weissenbacher, and co-authors, Bernhard Gleiss, Michael Rawson, Ahmed Bhayat, Clemens Eisenhofer and Martón Hajdu, for their expertise, intellectual exchange, and constructive feedback. On this note I'd also like to thank my reviewers Reiner Hähnle and Martina Seidl for their detailed reviews and constructive feedback of this thesis.

I am deeply appreciative of the support and camaraderie extended to me by my colleagues Petra, Sarah, Marcel, Sophie, Giovanni, Sanja, Federica, Anja, Michael and Matthias among many others at TU Wien. Your encouragement, insightful conversations, and friendship have been a constant source of enrichment. Special thanks also goes to Beatrix for your incredible support with all organizational and bureaucratic questions and concerns.

I owe a debt of gratitude to my parents, Brigitte and Nikolaos, particularly my mother, Brigitte, whose boundless encouragement, unwavering belief in my potential, and relentless support have been the cornerstone of my academic journey. Your encouragement to pursue my passions and dreams has been the driving force behind this success.

Furthermore, I extend my heartfelt appreciation to my grandparents, Hilde and Günther, for their unwavering support throughout my education. Your generosity and belief in my abilities have alleviated the burdens associated with academic pursuits, allowing me to focus wholeheartedly on my studies.

Special thanks are extended to my partner, Stefan, whose love, encouragement, and continuous confidence in my abilities have sustained me through the highs and lows of this pursuit, especially in the finishing stretches of writing it down.

To my dear friends who have accompanied me on this academic journey, especially Andjela, Corinna, Sandra and Sophie, your support and encouragement have been a source of strength during both the triumphs and challenges of doctoral studies. Your presence has made this experience more fulfilling and memorable.

Finally, to all those whose names may not be mentioned but whose support, encouragement, and contributions have played a significant role in shaping this thesis, I offer my sincere gratitude. This work would not have been possible without the collective support, encouragement, and guidance of all those mentioned above. Thank you for being an integral part of this transformative journey.

Kurzfassung

Diese Arbeit untersucht die Automatisierung induktiver Schlüsse für Programmverifikation mit Hilfe von automatisierten Theorembeweisern in der Prädikatenlogik erster Stufe, die auf dem sogenannten Superpositionskalkül basieren. Dabei untersuchen wir in erster Linie jene Programme, die Schleifen, Arrays oder rekursive Funktionsaufrufe enthalten. Wir schlagen neue Methoden zur Automatisierung induktiver Beweisführung in *theorem proving* vor, die die vollautomatische Verifikation solcher Programme erlaubt.

Im ersten Teil der Arbeit erforschen wir die Induktion in der sogenannten Trace Logic, einer Instanz der Prädikatenlogik erster Stufe mit Theorien und Datenstrukturen. Wir schlagen zwei Methoden vor, um induktive Schlüsse für Programmschleifen in Trace Logic handzuhaben und implementieren unsere Arbeit im Rapid Verifikationssystem und dem ihm zugrundeliegenden Theorem Prover Vampire.
In unserem ersten Ansatz erweitern wir die Programmsemantik in Trace Logic um sogenannte Trace Lemmata. Trace Lemmata drücken allgemeine, induktive Eigenschaften über Programme aus, die Schleifen, Integer und (unendliche) Arrays beinhalten können. Wir identifizieren eine Reihe sinnvoller Trace Lemmata, die den automatisierten Theorembeweiser beim induktiven Schließen unterstützen. Dies ermöglicht die vollständige Automatisierung von Beweisen durch beschränkte Induktion über Programmzeitpunkte. Darüber hinaus, erforschen wir induktive Inferenzen für Trace Logic direkt im zugrundeliegenden Theorem Prover um die Abhängigkeit von Trace Lemmata zu reduzieren. Hierfür erweitern wir den automatisierten Prover um zwei induktive Inferenzen, die spezifisch für Korrektheitsbeweise in Trace Logic verwendet werden, nämlich *multi-clause goal induction* und *array mapping induction*. Diese ermöglichen lemmaloses, induktives Beweisen gültiger Eigenschaften über Programme mit Schleifen, Arrays und ganzen Zahlen, ohne dass a priori induktive Lemmata oder Invarianten formalisiert werden müssen.

Der zweite Teil der Arbeit befasst sich mit Programmsemantik und induktiven Schlüssen rekursiver Programme. Insbesondere formalisieren wir die funktionale Programmsemantik in der Prädikatenlogik erster Stufe und zeigen unseren Ansatz, indem wir die Korrektheit gängiger Sortieralgorithmen beweisen. Zu diesem Zweck erweitern wir den Prover Vampire um spezifische, strukturelle Induktionsregeln basierend auf Listen, die durch einen beliebigen Datentyp parametrisiert sind, der eine lineare Ordnung zulässt. Wir liefern einen vollautomatischen Korrektheitsbeweis für den rekursiven `Quicksort`-Algorithmus unter anderen Sortieralgorithmen.

Abstract

This thesis explores automating inductive reasoning for software verification of programs containing loops, arrays and recursive function calls. We propose new methods of automating induction in first-order theorem proving based on the superposition calculus allowing for fully automated verification using theorem proving for such programs.

In the first part of the thesis, we explore *induction in trace logic*, an instance of many-sorted first-order logic with theories. We propose two methodologies to handle inductive loop reasoning in trace logic and implement our work in the Rapid verification framework and the underlying saturation prover Vampire. In our first approach, we extend trace logic program semantics with so-called trace lemmas. Trace lemmas express common properties over programs with loops, integers and unbounded arrays based on bounded induction over timepoints. We identify a set of trace lemmas for such programs. These lemmas help the automated theorem prover with inductive reasoning and enable the full automation of proofs of such programs. Furthermore, we reduce reliance on trace lemmas by introducing bounded induction over timepoints directly in the underlying theorem prover. That is, we extend the saturation-based prover with two inductive inferences specific to reasoning in trace logic, namely multi-clause goal induction and array mapping induction, for lemmaless reasoning over loop iterations. Both inference rules enable the prover to derive safety properties over programs with loops, arrays and integers without the need of a priori trace lemma reasoning.

The second part of the thesis deals with program semantics and *inductive reasoning for recursive programs*. Specifically, we formalize functional program semantics in many-sorted first order logic and showcase our approach by proving common sorting algorithms correct. To this end, we extend the saturation prover Vampire with specific structural induction rules based on lists parameterized by any sort that allows for a linear order, namely computation induction. Based on this methodology we provide a fully automated correctness proof of the recursive `Quicksort` algorithm among other sorting algorithms.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Over the course of the last few decades, the digital transformation of society has marked the world sustainably. In today's world, software is ubiquitous and has become an integral part of our day-to-day lives. From the first sound of our alarm clock to the precision of our GPS navigation system to the ease of our social media connections, software has permeated every aspect of life. However, its influence extends beyond individual interactions, besieging industries, health care, education, leisure, and more.
As technology advances, the incorporation of software into our daily lives not only provides convenience and efficiency, but can also put us at risk. Software bugs, often unexpected glitches or bugs in the code, come in all shapes and sizes, from minor hiccups to major malfunctions. Despite developers' best efforts to create robust, reliable software, the complexity of modern apps and the ever-evolving nature of technology make it difficult to eliminate bugs completely.
However, vulnerabilities can be costly. Several famous software bugs have made headlines over the years, causing significant disruptions, for example, at airports [Sko20], financial repercussions such as [Kan18, O'K20, She20] and breaches to personal data, most notably in social networks as for example [WS20, Rot22]. Whether it's the failure of a space mission, the billions of euros lost in financial transactions, or the disruption of communication networks that bring society to a standstill ([GNT10, WLL17, Kra22]) – these incidents demonstrate the significance of software quality assurance and the imperative of verifiable correctness of critical infrastructure in software development. More importantly, the ever-evolving nature of software necessitates the automatization of quality assurance processes.

Software Testing. Testing is the most widely used method of ensuring software quality and reducing the amount of errors that can lead to software failure or exploitation by

malicious attackers. Essentially, testing allows to check whether a computation matches the expectation for specific inputs and outputs.
From unit tests, white-box and black-box testing to automated test case generation and test coverage assessments, testing has become an integral part of the software development cycle with a vast range of methodologies, see for example [AO16, MBTS04]. Software developers are generally familiar with testing as part of the development process as most programming languages and frameworks come equipped with several tools that allow developers to find bugs. Due to this accessibility and practicability, software testing has evolved as the standard methodology to ensure software quality.
While many important software bugs can be found and circumvented with testing, full *correctness*, however, cannot be guaranteed. Granted, considerable progress has been made in the field of testing technology, both in the academic and industrial fields. Nevertheless, testing is often insufficient for asserting the safe execution of a program. As Dijkstra famously put it in [DDH72], "Program testing can be used to show the presence of bugs, but never to show their absence!". Given the profound integration of software in everyday life, Dijkstra's claim that a programmer's challenge is not only to implement a program but also show its correctness dating back to 1969 is as relevant as ever. We need methods to *verify* program correctness and, ideally, we want those approaches to be automated.

1.1.1 Program Correctness and Induction

Software verification is a robust methodology that eliminates the drawbacks of testing. The approach relies on verifying the absence of implementation errors by considering the general execution of a program, that is any computation independent of specific inputs. Doing so, its aim is to determine that a program meets specified (functional) behavior. The only way to reliably assert whether a software system meets the functional requirements for a particular piece of code is to specify and prove it – *formally* and *rigorously*. It is, thus, evident that in order to determine *program correctness*, mathematical proof is required: only a formal specification of program semantics and requirements allows us to establish the validity of safety properties. Proofs are the core of formal methods for software verification.

When it comes to mathematical proofs for program correctness one technique is indispensable: *induction*. The principle of mathematical induction allows us to elegantly prove general statements P, for instance, about natural numbers:

$$(P(0) \land \forall k.\, P(k) \to P(k+1)) \to \forall k.\, k \geq 0 \to P(k)$$

We first prove that P holds for the smallest natural number 0, the *base case* $P(0)$. We proceed to show the *step case* $\forall k.\, P(k) \to P(k+1)$ by assuming that P holds for some arbitrary but fixed number k. This assumption is called the *induction hypothesis* and it can be used to show that P still holds for $k+1$. Proving the base and the step case allows us to use the conclusion of the principle, that is P holds for all natural numbers.

In program verification, this proof concept is vital once our programs contain loops, algebraic data types, or recursive calls. It is not surprising that any system that evaluates large (and potentially infinite) numbers of possible executions of a software program utilizes induction, either implicitly or explicitly. When considering loops, it is necessary to provide an abstraction of each individual run and to demonstrate a general statement about what a loop can compute. This is achieved through the use of *loop invariants* which, to a certain degree, describe the behavior of a loop and abstract it from specific variable values.
The concept of loop invariants dates back to the 1960s when Hoare and Floyd [Flo67, Hoa69] began to recognize the significance of "assigning meaning to programs", that is conceptualizing them as mathematical objects that could be formally reasoned about. They proposed an axiomatic semantics and a deductive system for program correctness that is now widely known as Floyd-Hoare Logic and was further developed into the predicate transformers by the Dijkstra in 1975 [Dij75]. To this day, these works were the first systematic approaches to manual program verification and laid the theoretical foundation for automated verification procedures yet to come.

All of these works underpin how loop invariants are necessary for proving correctness: by showing that an invariant is *inductive* and that it implies a safety property (together with a negated loop condition), we can prove partial correctness. To demonstrate an invariant's inductive nature, we have to prove two things: (1) a loop invariant must hold prior to the first iteration, while (2) an invariant that holds prior to some iteration should also hold prior to the next iteration. In this context, (1) can be understood as showing the base case of an induction axiom, while condition (2) corresponds to proving the inductive step of an induction axiom.
However, writing such proofs by hand is tedious, time-consuming and not feasible for large amounts of code given the rapid evolution of code, even for security-sensitive applications. Hence, automation of (inductive) proof writing is indispensable for the verification process.

1.1.2 Automated Verification Techniques

Automating inductive proofs is notoriously hard:

> "I shall show that there is no general method which tells whether a given formula $\mathcal{U}$ is provable in **K**."

With these words from [Tur36], Alan Turing opened his famous proof showing that the Halting problem has no solution, that it is *undecidable*. His proof essentially eliminated all hope to find an automated procedure that when given a program and a property can decisively answer true or false. Some decision problems are undecidable. Together with Church's theorem [Chu36] stating that the set of all valid formulas of first-order logic is not effectively decidable, or Rice's theorem [Ric53] that any non-trivial property of a program is effectively undecidable, the outlook on an automated procedure to prove programs correct is at best grim.

However, not all hope is lost. The quest for reliable and rigorous methods to ensure the correctness of software has been a driving force in the evolution of automated verification techniques. The field of automated software verification, while proven to be deeply undecidable on multiple occasions, has experienced major advancements and has been shown to be effective in practice. Over the years, researchers and practitioners have explored various approaches, each contributing to the intricate tapestry of software verification [HH19, DKW08, WLBF09, APS14, KG99].

Satisfiability Modulo Theories. When it comes to automated reasoning techniques, *satisfiability modulo theories* (SMT) solvers based on the DPLL(T) decision procedure [Nel80, NO79] combining first-order logic and theories are on the forefront of automated technologies. Today, they build the reasoning engine for many different techniques of proving program correctness, such as [FPMG19, GSV18, CGU20, CGU21, Lei10] among many. Most notably Z3 [DMB08] and CVC4 [BCD+11] are used as deductive backends that come with strong reasoning for multiple theories and first-order logic. However, they have limitations in quantified reasoning as such solvers depend on quantifier instantiation strategies [DNS05, FJS04, GBT07, RTDM14] and are thus mostly restricted to quantifier-free and universally quantified formulas.
Additionally, the main challenge of handling inductive reasoning over loops in a first-order setting is still not solved in a clear-cut way. Many tools and approaches relying on SMT-solvers still depend on the user to specify loop invariants, contracts or other methods to annotate a program with inductive properties that can be used in automated verification to deduce whether a property holds, see e.g. [Lei10, FP13, PMP+14, ABB+05]. Others, while inferring loop invariants in an automated way, are restricted to quantifier-free or universally quantified properties and program semantics [FPMG19, GSV18, CGU20, CGU21].

Proof assistants. Interactive proof assistants based on higher-order logic do support inductive proofs. Provers such as `Isabelle/HOL` [NWP02], `Coq` [BC13], `ACL2` [BM90, KMM00] or the `KeY` system [ABB+05] allow to verify inductive proof steps in a machine-assisted way. However, while some heuristics are in place [BSVH+93] to determine which variables to use in induction, such tools usually rely on the proof engineer to decipher the correct way of applying induction during proof construction. That is, the user has to define or choose the induction scheme to apply when reasoning about loops or recursive function calls.

All of the above mentioned methodologies to reason about programs are based on induction, either explicitly or implicitly. Either an induction scheme must be explicitly provided to the prover, loop invariants are employed to reason about loop execution, or method contracts serve as an inductive hypothesis for recursive calls.

Superposition-based Theorem Proving. In contrast to the above approaches, first-order theorem provers [KV13] enable quantified reasoning modulo theories [KRV17, RBSV16, RSV21], such as linear integer arithmetic and arrays, in a fully automated

manner. The combination of full first order quantification, uninterpreted functions, and theory-specific symbols provides proof engineers with great modeling capabilities. Moreover, thanks to recent advances, they offer a possibility of built-in automated inductive reasoning [EP20, Cru15, HKV21, HHK+20, HHK+22]. First-order reasoning can, thus, complement the aforementioned verification efforts when it comes to proving program properties that require inductive reasoning and complex quantification.

In this thesis, we address the question of how to leverage and advance the development of first-order superposition-based theorem provers for the purpose of software verification and how to automate the inductive reasoning required for this aim.

1.2 Contributions

We believe that our work strongly advocates the use of automated first-order theorem provers with regards to software verification. Even in the sight of undecidability, inductive reasoning over programs containing recursive data structures, loops, linear arithmetic but also functional programming constructs can be efficiently automated by leveraging and extending their capabilities. Our contributions are summarized in the following.

Inductive Reasoning with Trace Logic. Trace logic $\mathcal{L}$ [BEG+19, GGK20a], an instance of many-sorted first-order logic, enables the partial correctness verification of imperative programs containing loops, arrays and linear integer arithmetic. Trace logic generalizes semantics of program locations and captures loop semantics by encoding properties at arbitrary timepoints and loop iterations. The crux in automating partial correctness proofs with saturation-based theorem proving is the automated handling of inductive reasoning over loops. We propose two different methodologies to handle inductive reasoning in trace logic and implement our work in the Rapid verification framework [GGB+22] and the underlying saturation prover Vampire [KV13]:

* *Trace Lemma Reasoning.* In our first approach [GGK20a] towards handling induction, we guide and automate inductive loop reasoning in trace logic $\mathcal{L}$ (Chapter 3). We automatically instantiate a set of predefined generic trace lemmas that represent common inductive properties over a wide set of programs containing loops, unbounded arrays and linear integer arithmetic. Intuitively, these lemmas capture inductive loop invariants over array-transforming loops with bounded induction over program execution timepoints.We prove soundness of each trace lemma.
* *Lemmaless Reasoning.* We extend trace logic with generic bounded induction schemata over timepoints and loop counters, reducing reliance on trace lemmas (Chapter 4). Inferring and proving loop invariants becomes an inductive inference step within superposition-based first-order theorem proving. We introduce two new inference rules, multi-clause goal induction and array mapping induction, for lemmaless reasoning over loop iterations. The inference rules are compatible with any saturation-based inference system used for first-order theorem proving and work by carrying out induction on terms corresponding to final loop iterations.

- *Invariant Extraction.* We revise symbol elimination and consequence-finding [KV09] for invariant extraction with first-order theorem provers in the context of trace logic $\mathcal{L}$ (Chapter 5).
- *The Rapid Verification Framework.* We implement our work in the Rapid framework for automatic software verification by applying first-order reasoning in trace logic (Chapter 7). Rapid establishes partial correctness of programs with loops and arrays by inferring invariants necessary to prove program correctness using a saturation-based automated theorem prover. Rapid can heuristically instantiate trace lemmas, or alternatively, exploit nascent support for induction to fully automate inductive reasoning in a lemmaless style. In addition, Rapid can be used as an invariant generation engine, supplying other verification tools with quantified loop invariants necessary for proving partial program correctness.

Computation Induction for Recursion. Apart from inductive reasoning in trace logic, we investigate built-in induction in saturation-based theorem proving to establish functional program correctness for recursive algorithms. Specifically we formalize functional program semantics in many-sorted first order logic and showcase our approach by proving common sorting algorithms correct. Full automation without requiring a priori defined invariants is powered by structural and recursion induction over *parameterized lists* in the superposition-based first-order theorem prover Vampire:

- *Recursive Sorting Algorithms.* We formalize the semantics of functional programs with recursive data structures in the first-order theory of lists with parameterized sorts (Chapter 6). Particularly, we capture the correctness of functional versions of common sorting routines via two properties over lists, namely the sortedness and the permutation equivalence property, and introduce a first-order formalization of these properties. Rather than focusing on specific first-order theories such as lists of integer arithmetic, our formalization relies on a parameterized sort abstracting (arithmetic) theories.
- *Computation Induction.* We further adjust recent efforts [HHK+20, HHK+22] for automating inductive reasoning in saturation-based first-order theorem proving. We extend first-order theorem proving to include inductive inferences based on computation induction. Based on our first-order semantics of sorting algorithms, we showcase compositional reasoning via first-order theorem provers with built-in induction.
- *Compositional Reasoning.* Importantly, we advocate a compositional reasoning approach for fully automating the verification of functional programs implementing and preserving sorting and permutation properties over parameterized list structures with saturation-based theorem proving. We exploit a divide-and-conquer approach implemented by sorting algorithms and provide a fully automated correctness proof of the recursive `Quicksort` algorithm. We generalize our inductive lemmas to prove the functional correctness of further recursive sorting algorithms such as `Mergesort` and `Insertionsort`.

1.3 Publications and Relation to Contributions

This thesis is based on the following publications:

[GGK20a] Pamina Georgiou, Bernhard Gleiss, and Laura Kovács. Trace Logic for Inductive Loop Reasoning. In *Proceedings of the 20th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2020)*, pages 255-263. TU Wien Academic Press, 2020.
The main content of Chapter 3 is based on this publication.

[BGE+22] Ahmed Bhayat, Pamina Georgiou, Clemens Eisenhofer, Laura Kovács, and Giles Reger. Lemmaless Induction in Trace Logic. In *International Conference on Intelligent Computer Mathematics (CICM 2022)* , pages 191-208. Springer, 2022.
Chapter 4 is extending this publication.

[GGB+22] Pamina Georgiou, Bernhard Gleiss, Ahmed Bhayat, Michael Rawson, Laura Kovács, and Giles Reger. The Rapid Software Verification Framework. In *Proceedings of the 22nd International Conference on Formal Methods in Computer-Aided Design (FMCAD 2022)*, pages 255-260. IEEE, 2022.
This publication serves as the foundation for Chapters 5 and 7.

[GHK23] Pamina Georgiou, Marton Hajdu, and Laura Kovács. Sorting Without Sorts. No. 10632. EasyChair Preprint, 2023. *Currently under submission.*
Chapter 6 draws upon the content of this publication.

For publications [BGE+22, GGB+22] and [GHK23], I acted as the main author, leading the respective research results of these papers. For [GGK20a] I have been the main author of efforts to guide inductive reasoning in trace logic.

Prior publication leading up to this thesis:

[BEG+19] Gilles Barthe, Renate Eilers, Pamina Georgiou, Bernhard Gleiss, Laura Kovács, and Matteo Maffei. Verifying relational properties using trace logic. In *Proceedings of the 19th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2019)*, pages 170-178. Springer, 2019

1.4 Outline

This thesis is organized as follows.

The first part of the thesis deals with automating inductive reasoning for trace logic $\mathcal{L}$ [GGK20a]. After giving some preliminaries on first-order theorem proving and program semantics in trace logic in Chapter 2, we dive into trace lemma reasoning in Chapter 3 based on [GGK20a]. Chapter 4 overviews our lemmaless reasoning approach [BGE+22] with built-in induction support for trace logic-based timepoint reasoning. Our invariant generation method for trace logic was first introduced in [GGB+22]. We revisit and extend our presentation based on symbol elimination in Chapter 5.

Chapter 6 deals with our built-in reasoning support of computation induction for recursive algorithms. We present our first-order formalizations of functional sorting routines and outline their compositional proofs based on our work in [GHK23].

In Chapter 7 we start by outlining the Rapid verification framework [GGB+22] with all its different capabilities based on trace lemma reasoning, lemmaless induction and invariant generation support. We proceed into our changes to the Vampire automated first-order theorem prover that supports our reasoning in trace logic, as well as computation induction for recursive algorithms in the theory of parameterized lists.

Finally, Chapter 8 summarizes our experiments drawing from multiple publications. We first describe our Rapid-based experimental evaluation of trace lemma versus lemmaless reasoning in Section 8.1. Additionally, we compare results with other state-of-the-art inductive reasoning tools. Furthermore, we report on our experiments on computation induction for sorting routines in Section 8.2 before concluding in Chapter 9.

CHAPTER 2

Background and Preliminaries

2.1 Saturation-based Automated Theorem Proving

Many-Sorted First-Order Logic. We consider standard many-sorted first-order logic with built-in equality, denoted by $\simeq$. We allow all standard boolean connectives and quantifiers of this language. By $s = F[u]$ we indicate that the term u is a subterm of s surrounded by (a possibly empty) context F.
We use x, y to denote variables, l, r, s, t for terms and sk for Skolem symbols. A *literal* is an atom A or its negation $\neg A$. A *clause* is a disjunction of literals $L_1 \vee ... \vee L_n$, for $n \geq 0$. A disjunction without literals, that is $n = 0$, is called the *empty clause* denoted by $\Box$. Given a formula F, we denote by $\mathrm{CNF}(F)$ the clausal normal form of F.
A *signature* is any finite set of symbols. We consider equality $\simeq$ as part of the language; hence, $\simeq$ is not a symbol. The signature of a formula F is the set of all symbols occurring in this formula. We write $F_1, \ldots, F_n \vDash F$ to denote that the formula $F_1 \wedge \ldots \wedge F_n \rightarrow F$ is a tautology. In particular, we write $\vDash F$, if F is valid.
For a logical variable x of sort S we write x_S. A *first-order theory* denotes the set of all valid formulas on a class of first-order structures. Any symbol in the signature of a theory is considered *interpreted*. All other symbols are *uninterpreted*. In particular, we use the theory of linear integer arithmetic denoted by $\mathbb{I}$ and the boolean sort $\mathbb{B}$. For a complete axiomatization of integer arithmetic and booleans, we refer to [RS17] and [KKV15] respectively.
We further consider the *theory of finite term algebras* for inductive data types. An inductive data type consists of a set of *constructors* of which at least one is constant. Let Σ be a finite set of function symbols (constructors) with at least one constant, consider

$$\bigvee_{f \in \Sigma} \exists y \,.\, x \simeq f(y). \tag{A1}$$

$$f(x) \not\simeq g(y) \tag{A2}$$

for every $f, g \in \Sigma$ such that $f \not\simeq g$.

$$f(x) \simeq f(y) \rightarrow x \simeq y \tag{A3}$$

for every $f \in \Sigma$ of arity ≥ 1.

$$t \not\simeq x \tag{A4}$$

for every non-variable term t in which x appears.

The set of formulas (A1) ot (A4) are the *axioms* of the theory of finite term algebras. Every element of Σ is equal to some purely inductive term (A1). Constructors are distinct (A2) and injective (A3). Terms are acyclic, that is no term is equal to its proper subterm (A4). For details of finite term algebras in the context of saturation based theorem proving, we refer to [KRV17].
We specifically use term algebras in two ways, that is for natural numbers and acyclic lists. We consider natural numbers as the term algebra $\mathbb{N}$ with four symbols in the signature: the constructors `0` and successor `suc`, as well as `pred` and $<$ respectively interpreted as the predecessor function and less-than relation. Note that we do not define any arithmetic on naturals. Further, we use the inductive datatype of lists with two constructors nil and $\mathsf{cons}(x, xs)$, where nil is the empty list and x and xs are respectively the head element and the recursive tail of a list.
We assume familiarity with the basics of saturation theorem proving, yet we provide a brief overview in the following.

Saturation. Rather than using arbitrary first-order formulas G, most first-order theorem provers rely on a clausal representation C of G. The task of first-order theorem proving is to establish that a formula/goal G is a logical consequence of a set $\mathcal{A}$ of clauses, including assumptions. Doing so, first-order provers clausify the negation $\neg G$ of G and derive that the set $S = \mathcal{A} \cup \{\neg G\}$ is unsatisfiable[1]. To this end, first-order provers *saturate* S by computing all logical consequences of S with respect to some sound inference system $\mathcal{I}$. A sound inference system $\mathcal{I}$ derives a clause D from clauses C such that $C \rightarrow D$. The saturated set of S w.r.t. $\mathcal{I}$ is called the *closure* of S w.r.t. $\mathcal{I}$, whereas the process of deriving the closure of S is called *saturation*. By soundness of $\mathcal{I}$, if the closure of S contains the empty clause $\square$, the original set S of clauses is unsatisfiable, implying the validity of $\mathcal{A} \rightarrow G$; in this case, we established a *refutation* of $\neg G$ from $\mathcal{A}$, hence a proof of validity of G.

Superposition Inference System. An *inference rule* is an n-ary relation on formulas, where $n \geq 0$. The elements of such a relation are called *inferences*, written as:

$$\frac{F_1 \quad \ldots \quad F_n}{\mathrm{F}}$$

[1] For simplicity, we denote by $\neg G$ the clausified form of the negation of G.

The formulas $F_1, \ldots, F_n$ are called the *premises* of this inference, whereas the formula F is the *conclusion* of the inference. An *inference system* is a set of inference rules. An *axiom* of an inference system is any conclusion of an inference with 0 premises.
In our work we use the *superposition inference system* [BG94, BG01, NR01], implemented by most modern automated theorem provers for full first-order logic. The *superposition calculus* is a common inference system used by saturation-based provers for FOL with equality.

The underlying superposition inference system is parameterized by a *simplification ordering* [DP01] such that it does not (for efficiency reasons) rewrite smaller terms into larger terms. For the sake of being self-contained, let us recall the notion of a *simplification orderings on terms*. An ordering $\succ$ on terms is a simplification ordering if the following four conditions hold:

1. $\succ$ is *well-founded*: there exists no infinite sequence of terms $t_0, t_1, ...$ such that $t_0 \succ t_1 \succ ....$
2. $\succ$ is *monotonic*: if $l \succ r$, then $s[l] \succ s[r]$, for all terms l, r, s.
3. $\succ$ is *stable under substitution*: if $l \succ r$, then $l\theta \succ r\theta$ for some substitution θ
4. $\succ$ has the *subterm property*: if l is a subterm of r and $l \neq r$, then $r \succ l$

A common simplification ordering in superposition-based theorem proving is the Knuth-Bendix ordering (KBO) [KB83], parameterized by a symbol precedence $\succ_S$ and a weight function w assigning weights to symbols. We denote KBO by $\succ_{kbo}$. The weight function allows to compare weights of terms and their subterms respectively if necessary.
Specifically, the weight function is lifted to terms as follows:

$$w(f(t_1, ..., t_n)) = w(f) + \sum w(t_i).$$

Let w_0 be a positive integer, such that $w(c) \geq w_0$ for all symbols c, where we have $w(x) = w_0$ for all variables x. Given terms s and t, we inductively define $s \succ_{kbo} t$ such that

(1) for all variables x, the number of occurrences of x in s is greater or equal to that in t, and
(2) either $w(s) > w(t)$,
(3) or $w(s) = w(t)$ and one of the following conditions holds:
   (a) t is a variable x and $s = f^n(x)$ for some function symbol f and $n > 0$
   (b) $s = f(s_1, ..., s_n), t = f(t_1, ..., t_n)$ where $s_i \succ_{kbo} t_i$ for some i and $s_j =_{kbo} t_j$ for $j < i$
   (c) $s = f(s_1, ..., s_n), t = g(t_1, ..., t_m)$ such that $f \succ_S g$.

Note that the symbol precedence $\succ_S$ can be defined in numerous ways, for instance by the number of occurrences in the problem or the arity of symbols such that symbols with smaller arity are also smaller with regards to the precedence.

While simplification orderings direct the rewriting order, a *literal selection function* [BG01] determines what terms to rewrite. For every non-empty clause, a selection function selects a non-empty subset of literals. That is, for a clause $\underline{L} \vee C$, literal L

is selected, denoted by underlining. Note that a selection function can select multiple (and even all) literals of a clause, for instance all maximal literals with respect to the simplification ordering. A standard complete selection function in the superposition calculus selects either a negative literal or all maximal literals with respect to $\succ$. We call such a selection function *well-behaved*.

While the superposition calculus is a family of inference systems parameterized by a simplification ordering and a selection function, we define a standard superposition inference system, denoted by SUP, as follows:

Binary Resolution

$$\frac{\underline{A} \vee C \quad \neg\underline{A'} \vee D}{(C \vee D)\theta}$$

where $\theta := \mathsf{mgu}(A, A')$.

Factoring

$$\frac{\underline{A} \vee \underline{A'} \vee C}{(A \vee C)\theta}$$

where $\theta := \mathsf{mgu}(A, A')$.

Superposition

$$\frac{\underline{s \simeq t} \vee C \quad L[s'] \vee D}{(L[t] \vee C \vee D)\theta} \qquad \frac{\underline{s \simeq t} \vee C \quad \underline{u[s'] \not\simeq u'} \vee D}{(u[t] \not\simeq u' \vee C \vee D)\theta} \qquad \frac{\underline{s \simeq t} \vee C \quad \underline{u[s'] \simeq u'} \vee D}{(u[t] \simeq u' \vee C \vee D)\theta}$$

where $\theta := \mathsf{mgu}(s, s')$; $t\theta \not\succeq s\theta$;
(first rule only) $L[s']$ is not an equality literal;
and (second and third rules only) $u'\theta \not\succeq u[s']\theta$.

Equality Resolution

$$\frac{\underline{s \not\simeq t} \vee C}{C\theta}$$

where $\theta := \mathsf{mgu}(s, t)$.

Equality Factoring

$$\frac{\underline{s \simeq t} \vee \underline{s' \simeq t'} \vee C}{(s \simeq t \vee t \not\simeq t' \vee C)\theta}$$

where $\theta := \mathsf{mgu}(s, s')$; $t\theta \not\succeq s\theta$; and $t'\theta \not\succeq t\theta$.

Figure 2.1: The superposition inference system SUP. Underlined literals are selected.

Given a well-behaved selection function, the superposition inference system SUP is *sound* and *refutationally complete*. For the former, we mean that if the empty clause $\square$ is derivable from a set S of formulas in SUP, then S is unsatisfiable. For the latter, we mean that for any unsatisfiable set S of formulas, saturation in SUP derives the empty clause $\square$ as a logical consequence of S.

2.2 Induction and Superposition

Inductive reasoning has recently been embedded in saturation-based theorem proving [HHK+22, HKV21, HHK+20], by extending the superposition calculus with a new inference rule based on *induction axioms*:

$$(\mathsf{Ind})\ \frac{\overline{L}[t] \vee C}{\mathsf{cnf}(\neg F \vee C)} \quad \text{where} \quad \begin{array}{l}(1)\ L[t] \text{ is a quantifier-free (ground) literal,}\\ (2)\ F \rightarrow \forall x.L[x] \text{ is a valid } \textit{induction axiom},\\ (3)\ \mathsf{cnf}(\neg F \vee C) \text{ is the clausal form of } \neg F \vee C.\end{array}$$

An *induction axiom* refers to an instance of a valid induction schema. In our work, we use different induction schemata, such as bounded induction over naturals (as defined in Section 3.2), structural and computational induction (Section 6.3) schemata.
In particular, we use the following *structural induction* schema over lists:

$$\Big(L[\mathsf{nil}] \wedge \forall x, ys.(L[ys] \rightarrow L[\mathsf{cons}(x, ys)])\Big) \rightarrow \forall zs.L[zs] \tag{2.1}$$

Then, considering the induction axiom resulting from applying schema (2.1) to L, we obtain the following Ind instance for lists:

$$\frac{\overline{L}[t] \vee C}{\begin{array}{c}\overline{L}[\mathsf{nil}] \vee L[\sigma_{ys}] \vee C\\ \overline{L}[\mathsf{nil}] \vee \overline{L}[\mathsf{cons}(\sigma_x, \sigma_{ys})] \vee C\end{array}}$$

where t is a ground term of sort list, $L[t]$ is ground, and σ_x and σ_{ys} are fresh constant symbols. The above Ind instance yields two clauses as conclusions and is applied during the saturation process.

2.3 Trace Logic

In this section we introduce trace logic, denoted by $\mathcal{L}$, as an instance of many-sorted first-order logic with theories. Trace logic is the logical basis for reasoning about software correctness with first-order theorem provers. Precisely, it is the logic that enables our approach to automated inductive reasoning for while-like languages, namely by *trace lemma reasoning* as will be introduced in Chapter 3 as well as *lemmaless reasoning* (Chapter 4). We thus, first introduce a while-like language $\mathcal{W}$ in Section 2.3.1, proceed with expressions in trace logic $\mathcal{L}$ (Section 2.3.2) and finally establish the axiomatic semantics of $\mathcal{W}$ in $\mathcal{L}$ in Section 2.4.

2.3.1 Programming Model $\mathcal{W}$

We consider programs written in an imperative while-like programming language $\mathcal{W}$. This section recalls terminology from [BEG+19], however adapted to our setting of safety verification as in [GGK20a]. Unlike [BEG+19], we do not consider multiple program traces in $\mathcal{W}$. Moreover, we extend [BEG+19] by defining our programming model for

```
 1  func main() {
 2    const Int[] a;
 3    Int[] b;
 4    Int i = 0;
 5    Int j = 0;
 6    while (i < a.length) {
 7      if (a[i] ≥ 0) {
 8        b[j] = a[i];
 9        j = j + 1:
10      }
11      i = i + 1;
12    }
13  }
14  assert(∀k_𝕀.∃l_𝕀.((0 ≤ k < j ∧ a.length ≥ 0) → b(k) = a(l)))
15
```

Figure 2.2: Program copying positive elements from array `a` to `b`. The safety property ensures that for any element in array `b`, there exists an equivalent element in `a`.

$$
\begin{aligned}
\text{program} &::= \text{function} \\
\text{function} &::= \texttt{func main()\{} \text{ subprogram } \texttt{\}} \\
\text{subprogram} &::= \text{statement} \mid \text{context} \\
\text{context} &::= \text{statement; ... ; statement} \\
\text{statement} &::= \text{atomicStatement} \\
&\quad \mid \texttt{if(} \text{ condition } \texttt{)\{} \text{ context } \texttt{\}} \texttt{ else } \texttt{\{} \text{ context } \texttt{\}} \\
&\quad \mid \texttt{while(} \text{ condition } \texttt{)\{} \text{ context } \texttt{\}}
\end{aligned}
$$

Figure 2.3: Grammar of $\mathcal{W}$.

arbitrarily nested programs. In Section 2.4, we then introduce a generalized program semantics in trace logic $\mathcal{L}$.
Figure 2.3 shows the (partial) grammar of our programming model $\mathcal{W}$, emphasizing the use of contexts to capture lists of statements. An input program in $\mathcal{W}$ has a single `main`-function, with arbitrary nestings of if-then-else conditionals and while-statements. For simplicity, whenever we refer to loops, we mean while-statements. We consider *mutable and constant variables*, where variables are either integer-valued numeric variables or arrays of such numeric variables. We include standard *side-effect free expressions over booleans and integers*.

2.3.2 Expressions in Trace Logic

Locations and Timepoints. A program in $\mathcal{W}$ is considered as sets of locations, with each location corresponding to positions/lines of program statements in the program.

Given a program statement $\mathtt{s}$, we denote by l_s its (program) location. We reserve the location l_{end} to denote the end of a program. For programs with loops, some program locations might be revisited multiple times. We therefore model locations l_s corresponding to a statement $\mathtt{s}$ in a loop as functions of *iterations* when the respective location is visited. For simplicity, we write l_s also for the functional representation of the location l_s of $\mathtt{s}$. We thus consider locations as timepoints of a program and treat them as being functions l_s over iterations. The target sort of locations l_s is $\mathbb{L}$. For each enclosing loop of a statement $\mathtt{s}$, the function symbol l_s takes arguments of sort $\mathbb{N}$, corresponding to loop iterations. Further, when $\mathtt{s}$ is a loop itself, we also introduce a function symbol nl_s with argument and target sort $\mathbb{N}$; intuitively, nl_s corresponds to the last loop iteration of $\mathtt{s}$, that is the first iteration such that the loop condition of $\mathtt{s}$ is false. We parameterize nl_s by an argument of sort $\mathbb{N}$ for each enclosing loop of $\mathtt{s}$. This way, nl_s denotes the iteration in which $\mathtt{s}$ terminates for given iterations of the enclosing loops of $\mathtt{s}$. We denote the set of all function symbols l_s as $S_{\mathbb{L}}$, whereas the set of all function symbols nl_s is written as S_{nl}.

Example 1 (Timepoints)**.** We refer to program statements $\mathtt{s}$ by their (first) line number in Figure 2.2. Thus, l_4 encodes the timepoint corresponding to the first assignment of $\mathtt{i}$ in the program (line 4). We write $l_6(\mathtt{0})$ and $l_6(n_6)$ to denote the timepoints of the first and last loop iteration, respectively. The timepoints $l_7(\mathtt{suc}(\mathtt{0}))$ and $l_7(it)$ correspond to the beginning of the loop body in the second and the it-th loop iterations, respectively. Note that we model natural numbers with term algebra expressions of $\mathbb{N}$.

Expressions over Timepoints. We next introduce commonly used expressions over timepoints. For each while-statement $\mathtt{w}$ of $\mathcal{W}$, we introduce a function $it^{\mathtt{w}}$ that returns a unique variable of sort $\mathbb{N}$ for $\mathtt{w}$, denoting loop iterations of $\mathtt{w}$. Let $w_1, \ldots, w_k$ be the enclosing loops for statement $\mathtt{s}$ and consider an arbitrary term it of sort $\mathbb{N}$. We define $tp_{\mathtt{s}}$ to be the expressions denoting the timepoints of statements $\mathtt{s}$ as

$$\begin{aligned} tp_{\mathtt{s}} &:= l_s(it^{w_1}, \ldots, it^{w_k}) && \text{if } \mathtt{s} \text{ is non-while statement} \\ tp_{\mathtt{s}}(it) &:= l_s(it^{w_1}, \ldots, it^{w_k}, it) && \text{if } \mathtt{s} \text{ is while-statement} \\ nl_{\mathtt{s}} &:= n_s(it^{w_1}, \ldots, it^{w_k}) && \text{if } \mathtt{s} \text{ is while-statement} \end{aligned}$$

If $\mathtt{s}$ is a while-statement, we also introduce $nl_{\mathtt{s}}$ to denote the last iteration of $\mathtt{s}$. Further, consider an arbitrary subprogram $\mathtt{p}$, that is, $\mathtt{p}$ is either a statement or a context. The timepoint $start_{\mathtt{p}}$ (parameterized by an iteration of each enclosing loop) denotes the timepoint when the execution of $\mathtt{p}$ has started and is defined as

$$start_{\mathtt{p}} := \begin{cases} tp_{\mathtt{p}}(\mathtt{0}) & \text{if } \mathtt{p} \text{ is while-statement} \\ tp_{\mathtt{p}} & \text{if } \mathtt{p} \text{ is non-while statement} \\ start_{\mathtt{s}_1} & \text{if } \mathtt{p} \text{ is context } \mathtt{s}_1;\ldots;\mathtt{s}_k \end{cases}$$

We also introduce the timepoint $end_{\mathtt{p}}$ to denote the timepoint upon which a subprogram p has been completely evaluated and define it as

$$end_{\mathtt{p}} := \begin{cases} start_{\mathtt{s}} & \text{if s occurs after p in a context} \\ end_{\mathtt{c}} & \text{if p is last statement in context c} \\ end_{\mathtt{s}} & \text{if p is context of if-branch or} \\ & \text{else-branch of s} \\ tp_{\mathtt{s}}(\mathtt{suc}(it^s)) & \text{if p is context of body of s} \\ l_{end} & \text{if p is top-level context} \end{cases}$$

Finally, if s is the topmost statement of the top-level context in `main()`, we define

$$start := start_s.$$

Program Variables. We express values of program variables v at various timepoints of the program execution. To this end, we model (numeric) variables v as functions $v : \mathbb{L} \mapsto \mathbb{I}$, where $v(tp)$ gives the value of v at timepoint tp. For array variables v, we add an additional argument of sort $\mathbb{I}$, corresponding to the position where the array is accessed; that is, $v : \mathbb{L} \times \mathbb{I} \mapsto \mathbb{I}$. The set of such function symbols corresponding to program variables is denoted by S_V.
Our framework for constant, non-mutable variables can be simplified by omitting the timepoint argument in the functional representation of such program variables, as illustrated below.

Example 2 (Program variables)**.** For Figure 2.2, we denote by $i(l_4)$ the value of program variable i before being assigned in line 4. As the array variable a is non-mutable (specified by `const` in the program), we write $a(i(l_7(it)))$ for the value of array a at the position corresponding to the current value of i at timepoint $l_7(it)$. For the mutable array b, we consider timepoints where b has been updated and write $b(l_8(it), j(l_8(it)))$ for the array b at position j at the timepoint $l_8(it)$ during the loop.

We emphasize that we consider (numeric) program variables v to be of sort $\mathbb{I}$, whereas loop iterations it are of sort $\mathbb{N}$. This is due to the fact, that reasoning over iterations is limited to reasoning over $\{0, \mathtt{suc}, \leq\}$ and we don't consider arithmetic expressions over addition and multiplication $\{+, *\}$ for loop iterations.

Program Expressions. Arithmetic constants and program expressions are modeled using integer functions and predicates. Let e be an arbitrary program expression and write $[\![\mathtt{e}]\!](tp)$ to denote the value of the evaluation of e at timepoint tp.
We continue by defining some properties over values of expressions e at arbitrary timepoints that we will use in the axiomatization of program statements in the following section: Let $v \in S_V$, that is a function v denoting a program variable v. Consider

$\mathtt{e}, \mathtt{e}_1, \mathtt{e}_2$ to be program expressions and let tp_1, tp_2 denote two timepoints. We define

$$Eq(v, tp_1, tp_2) := \begin{cases} \forall pos_{\mathbb{I}}.\ v(tp_1, pos) \simeq v(tp_2, pos), & \text{if } \mathtt{v} \text{ is an array} \\ v(tp_1) \simeq v(tp_2), & \text{otherwise} \end{cases}$$

to denote that the program variable $\mathtt{v}$ has the same values at tp_1 and tp_2. We further introduce

$$EqAll(tp_1, tp_2) := \bigwedge_{v \in S_V} Eq(v, tp_1, tp_2)$$

to define that all program variables have the same values at timepoints tp_1 and tp_2.

Remark 1 (Framing)**.** Note that we have to introduce these predicates as a result of having timepoints in our formalism: when an assignment at a timepoint occurs, we also have to exclude the possibility of a value change of other program variables not having been affected by some program statement. This closely relates to the *frame problem* in many first-order and temporal logics [MH69]. The frame problem in first-order logic essentially deals with the question of how to formalize things that do not happen upon an event. Since a theorem prover cannot infer by itself what does not happen when moving in time from one program line to the next, we have to formally exclude value changes to program variables that have not been affected by a specific program statement. The resulting *Eq* and *EqAll* predicates in our formalization can, thus, be understood as *frame axioms* that logically exclude the occurrences of any variable value changes by moving from timepoint tp_1 to timepoint tp_2.

We further define

$$Update(v, e, tp_1, tp_2) := v(tp_2) \simeq [\![\mathtt{e}]\!](tp_1) \wedge \textstyle\bigwedge_{v' \in S_V \setminus \{v\}} Eq(v', tp_1, tp_2),$$

asserting that the numeric program variable $\mathtt{v}$ has been updated while all other program variables $\mathtt{v'}$ remain unchanged. This definition is further extended to array updates as

$$\begin{aligned} &UpdateArr(v, e_1, e_2, tp_1, tp_2) := \\ &\quad \forall pos_{\mathbb{I}}.\ (pos \not\simeq [\![e_1]\!](tp_1) \rightarrow v(tp_2, pos) \simeq v(tp_1, pos)) \\ &\quad \wedge\ v(tp_2, [\![e_1]\!](tp_1)) \simeq [\![e_2]\!](tp_1) \\ &\quad \textstyle\bigwedge_{v' \in S_V \setminus \{v\}} Eq(v', tp_1, tp_2), \end{aligned}$$

to declare that the numeric array variable $\mathtt{v}$ is updated at timepoint tp_2 at the position given by the evaluation of expression $[\![e_1]\!](tp_1)$ to the value of $[\![e_2]\!](tp_1)$ while array contents in all other positions remain unchanged.

Example 3 (Updates)**.** In Figure 2.2, we refer to the value of `i+1` at timepoint $l_{11}(it)$ as $i(l_{11}(it)) + 1$. Let S_V^1 be the set of function symbols representing the program variables of Figure 2.2. For an update of `j` in line 9 at some iteration it, we derive

$$\begin{aligned} \mathit{Update}(j, \texttt{j+1}, l_8(it), l_9(it)) := j(l_9(it)) \simeq (j(l_8(it)) + 1) \\ \wedge \bigwedge_{v' \in S_V^1 \setminus \{j\}} Eq(v', l_8(it), l_9(it)). \end{aligned}$$

For the array update of `b` in line 8, we have

$$\begin{aligned} &\mathit{UpdateArr}(b, \texttt{j}, \texttt{a[i]}, l_7(it), l_8(it)) := \\ &\quad \forall pos_{\mathbb{I}}.\ (pos \not\simeq j(l_7(it)) \rightarrow b(l_8(it), pos) \simeq b(l_7(it), pos) \\ &\quad \wedge\ b(l_8(it), j(l_7(it))) \simeq a(i(l_7(it))) \\ &\quad \textstyle\bigwedge_{v' \in S_V \setminus \{b\}} Eq(v', l_7(it), l_8(it)). \end{aligned}$$

2.4 Axiomatic Semantics in Trace Logic $\mathcal{L}$

Trace logic $\mathcal{L}$ has been introduced in [BEG+19], yet for the setting of relational verification. In this work we use the generalized formalization introduced in [GGK20a].

2.4.1 Trace Logic $\mathcal{L}$

Trace logic $\mathcal{L}$ is an instance of many-sorted first-order logic with equality. We define the signature $\Sigma(\mathcal{L})$ of trace logic as

$$\Sigma(\mathcal{L}) := S_{\mathbb{N}} \cup S_{\mathbb{I}} \cup S_{\mathbb{L}} \cup S_V \cup S_n,$$

containing respectively the signatures of the theory of natural numbers (as a term algebra) $\mathbb{N}$, the in-built integer theory $\mathbb{I}$, as well the respective sets of symbols for timepoints $S_{\mathbb{L}}$, program variables S_V and last iterations S_n as defined in section 2.3.2.
We next define the semantics of $\mathcal{W}$ in trace logic $\mathcal{L}$.

2.4.2 Reachability and its Axiomatization

We introduce a predicate $\mathit{Reach} : \mathbb{L} \mapsto \mathbb{B}$ to capture the set of timepoints reachable in an execution and use *Reach* to define the axiomatic semantics of $\mathcal{W}$ in trace logic $\mathcal{L}$. We define reachability *Reach* as a predicate over timepoints, in contrast to defining reachability as a predicate over program configurations such as in [HB12, BGMR15, FPMG19, ISIRS20]. We axiomatize *Reach* using trace logic formulas as follows.

Definition 1 (*Reach*-predicate)**.** *For any context c, any statement s, let Cond_s be the*

expression denoting a potential branching condition in s. *We define*

$$Reach(start_c) := \begin{cases} true, \\ \quad \textit{if } c \textit{ is top-level context} \\ Reach(start_s) \wedge Cond_s(start_s), \\ \quad \textit{if } c \textit{ is context of if-branch of } s \\ Reach(start_s) \wedge \neg Cond_s(start_s), \\ \quad \textit{if } c \textit{ is context of else-branch of } s \\ Reach(start_s) \wedge it^s < nl_s, \\ \quad \textit{if } c \textit{ is context of body of } s. \end{cases}$$

For any non-while statement s' *occurring in context* c, *let*

$$Reach(start_{s'}) := Reach(start_c),$$

and for any while-statement s' *occurring in context* c, *let*

$$Reach(tp_{s'}(it^{s'})) := Reach(start_c) \wedge it^{s'} \leq nl_{s'}.$$

Finally let $Reach(end) := true$.

Note that our reachability predicate *Reach* allows specifying properties about intermediate timepoints (since those properties can only hold if the referred timepoints are reached) and supports reasoning about which locations are reached.

2.4.3 Axiomatic Semantics of $\mathcal{W}$

We axiomatize the semantics of each program statement in $\mathcal{W}$, and define the semantics of a program in $\mathcal{W}$ as the conjunction of all these axioms.

Main-function. Let $\mathtt{p}_0$ be an arbitrary, but fixed program in $\mathcal{W}$; we give our definitions relative to $\mathtt{p}_0$. The semantics of $\mathtt{p}_0$, denoted by $[\![\mathtt{p}_0]\!]$, consists of a conjunction of one implication per statement, where each implication has the reachability of the start-timepoint of the statement as premise and the semantics of the statement as conclusion:

$$[\![\mathtt{p}_0]\!] := \bigwedge_{\mathtt{s} \text{ statement of } \mathtt{p}_0} \forall enclIts.(Reach(start_s) \rightarrow [\![\mathtt{s}]\!])$$

where *enclIts* is the set of iterations $\{it^{w_1}, \ldots, it^{w_n}\}$ of all enclosing loops $w_1, \ldots, w_n$ of some statement $\mathtt{s}$ in $\mathtt{p}_0$, and the semantics $[\![\mathtt{s}]\!]$ of program statements $\mathtt{s}$ is defined as follows.

Skip. Let $\mathtt{s}$ be a statement `skip`. Then

$$[\![\mathtt{s}]\!] := EqAll(end_{\mathtt{s}}, start_{\mathtt{s}}) \tag{2.2}$$

Integer assignments. Let s be an assignment v = e, where v is an integer-valued program variable and e is an expression. The evaluation of s is performed in one step such that, after the evaluation, the variable v has the same value as e before the evaluation. All other variables remain unchanged and thus

$$[\![\mathtt{s}]\!] := \mathit{Update}(v, e, \mathit{end}_s, \mathit{start}_s) \tag{2.3}$$

Array assignments. Consider s of the form $\mathtt{a[e_1]} = \mathtt{e_2}$, with a being an array variable and $\mathtt{e_1}, \mathtt{e_2}$ being expressions. The assignment is evaluated in one step. After the evaluation of s, the array a contains the value of $\mathtt{e_2}$ before the evaluation at position *pos* corresponding to the value of $\mathtt{e_1}$ before the evaluation. The values at all other positions of a and all other program variables remain unchanged and hence

$$[\![\mathtt{s}]\!] := \mathit{UpdateArr}(v, e_1, e_2, \mathit{end}_s, \mathit{start}_s) \tag{2.4}$$

Conditional if-then-else Statements. Let s be $\mathtt{if(Cond)\{c_1\}\ else\ \{c_2\}}$. The semantics of s states that entering the if-branch and/or entering the else-branch does not change the values of the variables and we have

$$[\![\mathtt{s}]\!] := \quad [\![\mathtt{Cond}]\!](\mathit{start}_s) \rightarrow \mathit{EqAll}(\mathit{start}_{\mathtt{c_1}}, \mathit{start}_s) \tag{2.5a}$$
$$\wedge \quad \neg[\![\mathtt{Cond}]\!](\mathit{start}_s) \rightarrow \mathit{EqAll}(\mathit{start}_{\mathtt{c_2}}, \mathit{start}_s) \tag{2.5b}$$

where the semantics $[\![\mathtt{Cond}]\!]$ of the expression Cond is according to Section 2.3.2.

While-Statements. Let s be the while-statement $\mathtt{while(Cond)\{c\}}$. We refer to Cond as the *loop condition*. The semantics of s is captured by conjunction of the following three properties: (2.6a) the iteration nl_s is the first iteration where Cond does not hold, (2.6b) entering the loop body does not change the values of the variables, (2.6c) the values of the variables at the end of evaluating s are the same as the variable values at the loop condition location in iteration nl_s. As such, we have

$$[\![\mathtt{s}]\!] := \quad \forall it^s_{\mathbb{N}}.\ (it^s < nl_{\mathtt{s}} \rightarrow [\![\mathtt{Cond}]\!](tp_{\mathtt{s}}(it^s)))$$
$$\wedge \quad \neg[\![\mathtt{Cond}]\!](tp(nl_{\mathtt{s}})) \tag{2.6a}$$
$$\wedge \quad \forall it^s_{\mathbb{N}}.\ (it^s < nl_{\mathtt{s}} \rightarrow \mathit{EqAll}(\mathit{start}_{\mathtt{c}}, tp_{\mathtt{s}}(it^s)) \tag{2.6b}$$
$$\wedge \quad \mathit{EqAll}(\mathit{end}_{\mathtt{s}}, tp_s(nl_{\mathtt{s}})) \tag{2.6c}$$

2.4.4 Soundness and Completeness.

The axiomatic semantics of $\mathcal{W}$ in trace logic is sound. That is, given a program p in $\mathcal{W}$ and a trace logic property $F \in \mathcal{L}$, we have that any interpretation in $\mathcal{L}$ is a model of F according to the small-step operational semantics of $\mathcal{W}$. We conclude with the next theorem - and refer to [GGK20b] for details.

Theorem 1 ($\mathcal{W}$-Soundness)**.** *Let p be a program. Then the axiomatic semantics $[\![p]\!]$ is sound with respect to standard small-step operational semantics.*

Next, we show that the axiomatic semantics of $\mathcal{W}$ in trace logic $\mathcal{L}$ is complete with respect to Hoare logic [Hoa69], as follows.
Intuitively, a Hoare Triple $\{F_1\}\mathtt{p}\{F_2\}$ corresponds to the trace logic formula

$$\forall enclIts.\big(Reach(start_p) \rightarrow ([F_1](start_p) \rightarrow [F_2](end_p))\big) \tag{2.7}$$

where the expressions $[F_1](start_p)$ and $[F_2](end_p)$ denote the result of adding to each program variable in F_1 and F_2 the timepoints $start_p$ respectively end_p as first arguments. We therefore define that the axiomatic semantics of $\mathcal{W}$ is *complete with respect to Hoare logic*, if for any Hoare triple $\{F_1\}\mathtt{p}\{F_2\}$ valid relative to the background theory $\mathcal{T}$, the corresponding trace logic formula (2.7) is derivable from the axiomatic semantics of $\mathcal{W}$ in the background theory $\mathcal{T}$. With this definition at hand, we get the following result, proved formally in [GGK20b].

Theorem 2 ($\mathcal{W}$-Completeness with respect to Hoare logic)**.** *The axiomatic semantics of $\mathcal{W}$ in trace logic is complete with respect to Hoare logic.*

CHAPTER 3

Trace Lemma Reasoning

A prior version of this chapter has been published as

> Pamina Georgiou, Bernhard Gleiss, and Laura Kovács. Trace Logic for Inductive Loop Reasoning. In *Proceedings of the 20th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2020)*, pages 255-263. TU Wien Academic Press, 2020.

One of the main challenges in automating software verification comes with handling inductive reasoning over programs containing loops. Until recently, automated reasoning in formal verification was the primary domain of satisfiability modulo theory (SMT) solvers [DMB08, BCD+11], yielding powerful advancements for inferring and proving loop properties with linear arithmetic and limited use of quantifiers, see e.g. [KBI+17, GSV18, FPMG19]. Formal verification, however, also requires reasoning about unbounded data types, such as arrays, and inductively defined data types in combination with full first-order quantification and (arithmetic) theories.

Specifying, for example as shown in Figure 3.1, that every element in the array `b` is initialized by a non-negative array element of `a` requires reasoning with quantifiers and can be best expressed in many-sorted extensions of first-order logic such as trace logic $\mathcal{L}$ (Section 2.3). Trace logic enables automated verification by expressing program semantics in $\mathcal{L}$ and using superposition-based first-order theorem proving to prove partial correctness of such programs.

However, using first-order reasoning with theories begs the question of how to handle inductive reasoning automatically in the presence of loops. In this chapter we will address the automation of induction by introducing *trace lemma reasoning*. In a nutshell, trace lemmas are helpful inductive properties over arbitrary program timepoints/loop iterations that enable the underlying reasoner to find proofs of partial software correctness in the superposition calculus. For programs such as Figure 3.1 that contain integers, arrays and loops, we identified a set of three such lemmas supporting the automated verification

```
 1  func main() {
 2    const Int[] a;
 3    Int[] b;
 4    Int i = 0;
 5    Int j = 0;
 6    while (i < a.length) {
 7      if (a[i] ≥ 0) {
 8        b[j] = a[i];
 9        j = j + 1:
10      }
11      i = i + 1;
12    }
13  }
14  assert(∀k_𝕀.∃l_𝕀.((0 ≤ k < j(end) ∧ a.length ≥ 0) → b(k, end) = a(l)))
15
```

Figure 3.1: Program copying positive elements from array `a` to `b`. The safety property here is expressed in trace logic $\mathcal{L}$. It ensures that for any element in array `b`, there exists an equivalent element in `a`.

process of multiple properties and programs that handle unbounded integer arrays with loops. We hence outline how trace lemmas are used to handle inductive reasoning steps for programs containing such programming constructs. To showcase the necessary reasoning steps, we will outline how trace lemmas are instantiated for two sample programs, namely Figure 3.1 and Figure 3.2.

3.1 Trace Logic for Safety Verification

Let us introduce the use of trace logic $\mathcal{L}$ for verifying safety properties of $\mathcal{W}$ programs and outline the use of trace lemma reasoning throughout this process. We consider safety properties F expressed in trace logic $\mathcal{L}$, as illustrated in line 14 of Figure 3.1. Thanks to soundness and completeness of the axiomatic semantics of $\mathcal{W}$, a partially correct program `p` with regard to F can be proved to be correct using the axiomatic semantics of $\mathcal{W}$ in trace logic $\mathcal{L}$. That is, we assume termination and establish partial program correctness. Assuming the existence of an iteration violating the loop condition can be help backward reasoning and, in particular, automatic splitting of loop iteration intervals.
However, proving correctness of a program `p` annotated with a safety property F faces the reasoning challenges of the underlying logic, in our case of trace logic. Due to the presence of loops in $\mathcal{W}$, a challenging aspect in using trace logic for safety verification is to handle inductive reasoning as induction cannot be generally expressed in first-order logic. To circumvent the challenge of inductive reasoning and automate verification using trace logic, we introduce a set of trace lemmas, and extend the semantics of $\mathcal{W}$ programs in trace logic with these trace lemmas. Trace lemmas describe generic inductive properties over arbitrary loop iterations and any logical consequence of trace lemmas yields a valid

```
 1  func main() {
 2    const Int[] a;
 3    Int[] b, c;
 4    Int i, j, k = 0, 0, 0;
 5    while(i < a.length) {
 6      if(a[i] ≥ 0) {
 7        b[j] = a[i];
 8        j = j + 1;
 9      } else {
10        c[k] = a[i];
11        k = k + 1;
12      }
13      i = i+1;
14    }
15  }
16  assert(∀k_𝕀.∃l_𝕀.((0 ≤ k < i(end) ∧ a.length ≥ 0 ∧ a(k) ≥ 0) → b(l, end) = a(k)))
17
```

Figure 3.2: Program partitioning an array `a` into two arrays `b, c` containing positive and negative elements of `a` respectively. The specification expresses that for every positive element in `a`, there exists an element in array `b` after the computation for some input array `a` of arbitrary non-negative length.

program loop property as well. We summarize our approach to program verification using trace logic $\mathcal{L}$:

Safety Verification in Trace Logic. Given a program $\mathtt{p}$ in $\mathcal{W}$ and a safety property F,

(i) we express program semantics $[\![\mathtt{p}]\!]$ in trace logic $\mathcal{L}$, as given in Section 2.4;

(ii) we formalize the safety property in trace logic $\mathcal{L}$, that is we express F by using program variables as functions of locations and timepoints (see assertion in Figure 2.2 and its translation to trace logic $\mathcal{L}$ in Figure 3.1). For simplicity, let us denote the trace logic formalization of F also by F;

(iii) we introduce instances $\mathcal{T}_{\mathcal{L}}^{\mathtt{p}}$ of a set $\mathcal{T}_{\mathcal{L}}$ of trace lemmas, by instantiating trace lemmas with program variables, locations and timepoints of $\mathtt{p}$;

(iv) to verify F, we then show that F is a logical consequence of $[\![\mathtt{p}]\!] \wedge \mathcal{T}_{\mathcal{L}}^{\mathtt{p}}$;

(v) however to conclude that $\mathtt{p}$ is partially correct with regard to F, two more challenges need to be addressed. First, in addition to Theorem 1, soundness of our trace lemmas $\mathcal{T}_{\mathcal{L}}$ needs to be established, implying that our trace lemma instances $\mathcal{T}_{\mathcal{L}}^{\mathtt{p}}$ are also sound. Soundness of $\mathcal{T}_{\mathcal{L}}^{\mathtt{p}}$ implies then validity of F, whenever F is proven to be a logical consequence of sound formulas $[\![\mathtt{p}]\!] \wedge \mathcal{T}_{\mathcal{L}}^{\mathtt{p}}$. However, to ensure that F is provable in trace logic, as a second challenge we need to ensure that our trace lemmas $\mathcal{T}_{\mathcal{L}}$, and thus their instances $\mathcal{T}_{\mathcal{L}}^{\mathtt{p}}$, are strong enough to prove $[\![\mathtt{p}]\!] \wedge \mathcal{T}_{\mathcal{L}}^{\mathtt{p}} \implies F$. That is, proving that F is a safety assertion of $\mathtt{p}$ in our setting requires finding a suitable set $\mathcal{T}_{\mathcal{L}}$ of trace lemmas.

3.2 A Suitable Set of Trace Lemmas for $\mathcal{W}$ programs

Finding a suitable set of trace lemmas heavily depends on the underlying programs and safety properties that are addressed. In this section, we introduce a set of three trace lemmas that were established to be strong enough to prove a wide range of example programs containing integers, unbounded integer-arrays and loops correct. We further show that these trace lemmas $\mathcal{T}_{\mathcal{L}}$ are sound consequences of bounded induction.

Bounded Induction over Loop Iterations. Let P be a first-order formula with one free variable x of sort $\mathbb{N}$. We recall the standard (step-wise) induction scheme for natural numbers as being

$$\Big(P(0) \wedge \forall x'_{\mathbb{N}}.(P(x') \rightarrow P(\mathtt{suc}(x')))\Big) \rightarrow \forall x_{\mathbb{N}}.P(x) \qquad (\mathit{Ind})$$

In our work, we use a variation of the induction scheme (Ind) to reason about intervals of loop iterations. Let P be an arbitrary trace logic formula with free variables bl and br, we use the following schema of *bounded induction*

$$\begin{aligned}&\Big(P(bl) \wedge && \text{(base case)}\\ &\quad \forall x'_{\mathbb{N}}.\Big((bl \leq x' < br \wedge P(x')) \rightarrow P(\mathtt{suc}(x'))\Big) && \text{(inductive case)}\\ &\rightarrow \forall x_{\mathbb{N}}.\Big(bl \leq x \leq br \rightarrow P(x)\Big),\end{aligned} \qquad (\mathit{B\text{-}Ind})$$

where $bl, br \in \mathbb{N}$ are term algebra expressions of $\mathbb{N}$, referred to respectively as left and right bounds of bounded induction.

Trace Lemmas $\mathcal{T}_{\mathcal{L}}$ for Verification. Trace logic properties support arbitrary quantification over timepoints and describe values of program variables at arbitrary loop iterations and timepoints. We therefore can relate timepoints with values of program variables in trace logic $\mathcal{L}$, allowing us to describe the value distributions of program variables as functions of timepoints throughout program executions. As such, trace logic $\mathcal{L}$ supports

(1) reasoning about the *existence* of a specific loop iteration, allowing us to split the range of loop iterations at a particular timepoint, based on the safety property we want to prove. For example, we can express and derive loop iterations corresponding to timepoints where one program variable takes a specific value for *the first time during loop execution*;

(2) universal quantification over the array content and range of loop iterations bounded by two arbitrary left and right bounds, allowing us to apply instances of the induction scheme ($\mathit{B\text{-}Ind}$) within a range of loop iterations bounded, for example, by it and nl_s for some while-statement $\mathtt{s}$.

To capitalize on these advantages of trace logic, we express generic patterns of inductive program properties as *trace lemmas*. Identifying a suitable set $\mathcal{T}_{\mathcal{L}}$ of trace lemmas to

automate inductive reasoning in trace logic $\mathcal{L}$ is however challenging and domain-specific. We propose three trace lemmas for inductive reasoning over arrays and integers, by considering

(A1) one trace lemma describing how values of program variables change during an interval of loop iterations;

(B1-B2) two trace lemmas to describe the behavior of loop counters.

(A1) Value Evolution Trace Lemma

Let w be a while-statement, let v be a mutable program variable and let $\circ$ be a reflexive and transitive relation - that is $\simeq$, $\leq$ or $\geq$ in the setting of trace logic. The *value evolution trace lemma of w, v, and* $\circ$ is defined as

$$\begin{aligned}&\forall bl_{\mathbb{N}}, br_{\mathbb{N}}.\\&\Big(\forall it_{\mathbb{N}}.\Big((bl \leq it < br \wedge v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(it)))\\&\qquad\qquad \rightarrow v(tp_{\mathtt{w}}(bl)) \circ v(tp_{\mathtt{w}}(\mathtt{suc}(it)))\Big)\\&\quad \rightarrow (bl \leq br \rightarrow v(tp_{\mathtt{w}}(br)) \circ v(tp_{\mathtt{w}}(br)))\Big)\end{aligned} \tag{A1}$$

In our work, the value evolution trace lemma is mainly instantiated with the equality predicate $\simeq$ to conclude that the value of a variable does not change during a range of loop iterations, provided that the variable value does not change at any of the considered loop iterations.

Example 4. For Figure 3.1, the value evaluation trace lemma (A1) yields the property

$$\begin{aligned}&\forall j_{\mathbb{I}}.\ \forall bl_{\mathbb{N}}.\ \forall br_{\mathbb{N}}.\\&\Big(\forall it_{\mathbb{N}}.\Big((bl \leq it < br\ \wedge\ b(l_6(bl), j) \simeq b(l_6(it), j))\\&\qquad\qquad \rightarrow b(l_6(bl), j) \simeq b(l_6(s(it)), j)\Big)\\&\rightarrow (bl \leq br \rightarrow b(l_6(bl), j) \simeq b(l_6(br), j))\Big),\end{aligned}$$

which allows to prove that the value of b at some position j remains the same from the timepoint it where the value was first set until the end of program execution. That is, we derive $b(l_6(end), j(l_6(it))) \simeq a(i(l_6(it)))$. Note that the lemma is instantiated with the timepoint l_6 that denotes the beginning of the loop.

Remark 2 (Quantified Bounds)**.** Effectively, in Example 4 the prover uses the value evolution lemma to conclude that from the next iteration after the value of $b(l_8(it), j(l_8(it)))$ is changed to $a(i(l_8(it)))$ for some iteration it the values of b at this position are not changed any further at any time point between and including the left bound bl and the right bound br. Specifically to derive $b(l_6(end), j(l_6(it))) \simeq a(i(l_6(it)))$, the prover infers that bl is $\mathtt{suc}(it)$ and br is nl_6. This begs the question why lemma bounds bl, br are

quantified and not immediately instantiated. The reason for this is the way proof search works in superposition-based theorem provers: the left bound mostly stems from the negation of the conjecture and is thus a skolem function. The lemma is hence simplified with this skolem to derive a contradiction, hence (given multiple other proof steps) proving the validity of the original property. Thus, choosing the right term, particularly for the left bound, has to be performed by the prover during superposition-based proof search.

Example 5. Similarly, (A1) is applied to prove the safety property of Figure 3.2:

$$\begin{aligned}
&\forall j_{\mathbb{I}}.\ \forall bl_{\mathbb{N}}.\ \forall br_{\mathbb{N}}.\\
&\Big(\forall it_{\mathbb{N}}.\Big((bl \leq it < br\ \wedge\ b(l_5(bl), j) \simeq b(l_5(it), j))\\
&\qquad \rightarrow b(l_5(bl), j) \simeq b(l_5(s(it)), j)\Big)\\
&\rightarrow (bl \leq br \rightarrow b(l_5(bl), j) \simeq b(l_5(br), j))\Big),
\end{aligned}$$

that is we can conclude that if for any iteration it bigger than or equal to the left bound bl, the value of b at some position j remains step-wise the same in any successive iteration $\texttt{suc}(it)$, we have that the values of b at position j remain unchanged throughout and including at the right bound bl.

Density

For the following two lemmas, we introduce the notion of *dense* integer variables. Let $\texttt{w}$ be a while-statement and let $\texttt{v}$ be a mutable program variable. We call $\texttt{v}$ to be *dense* if the following holds:

$$\begin{aligned}
Dense_{w,v} := \forall it_{\mathbb{N}}.\Big(&it < nl_{\texttt{w}} \rightarrow\\
&(v(tp_{\texttt{w}}(\texttt{suc}(it))) \simeq v(tp_{\texttt{w}}(it))\ \vee\\
&v(tp_{\texttt{w}}(\texttt{suc}(it))) \simeq v(tp_{\texttt{w}}(it)) + 1)\Big)
\end{aligned}$$

Further, we refer to $\texttt{v}$ as *strongly-dense* if the following holds:

$$\begin{aligned}
StrDense_{w,v} := \forall it_{\mathbb{N}}.\Big(&it < nl_{\texttt{w}} \rightarrow\\
&(v(tp_{\texttt{w}}(\texttt{suc}(it))) \simeq v(tp_{\texttt{w}}(it)) + 1)\Big)
\end{aligned}$$

Note that we use two variations of density with the following intuition. While some numeric program variables, particularly loop counters are incremented in each of the loop's iterations, that is they are *strongly dense*, others are only conditionally updated. Notably, program variables such as array length counters might be incremented based on branching, thus an increment might only occur in *some* of the loop iterations. Such variables are hence *dense*, but not *strongly-dense*.

(B1) Intermediate Value Trace Lemma

Let w be a while-statement and let v be a mutable program variable. The *intermediate value trace lemma of* w *and* v is defined as

$$\begin{aligned}\forall x_{\mathbb{I}}.\Big((Dense_{w,v} \wedge v(tp_{\mathtt{w}}(\mathtt{0})) \leq x < v(tp_{\mathtt{w}}(nl_{\mathtt{w}}))) \rightarrow \\ \exists it_{\mathbb{N}}.(it < nl_{\mathtt{w}} \wedge\ v(tp_{\mathtt{w}}(it)) \simeq x \wedge \\ v(tp_{\mathtt{w}}(\mathtt{suc}(it))) \simeq v(tp_{\mathtt{w}}(it)) + 1)\Big)\end{aligned} \tag{B1}$$

The intermediate value trace lemma (B1) allows us to conclude that if the variable v is dense, and if the value x is between the value of v at the beginning of the loop and the value of v at the end of the loop, then there exists an iteration in the loop, where v has exactly the value x and, particularly in this iteration is about to be incremented. This trace lemma is mostly used to find specific iterations corresponding to positions x in an array. Specifically, it enables the prover to determine the iteration where an update at some position of an array occurs.

Example 6. In Figure 3.1, using trace lemma (B1) we synthesize the iteration it such that $b(l_8(it), j(l_8(it))) \simeq a(i(l_8(it)))$.

$$\begin{aligned}\forall x_{\mathbb{I}}.\Big((Dense_{l_6,i} \wedge j(l_6(\mathtt{0})) \leq x < j(l_6(nl_6))) \rightarrow \\ \exists it_{\mathbb{N}}.(it < nl_6 \wedge\ j(l_6(it)) \simeq x \wedge \\ j(l_6(\mathtt{suc}(it))) \simeq j(l_6(it)) + 1)\Big).\end{aligned}$$

Given the semantics of the program in Figure 3.1, the prover can conclude that it is the iteration when the update to j, hence also to b at the current value of j, occurs.

Example 7. In Figure 3.2 we use this lemma to determine the iteration it where the update $b(l_7(it), j(l_7(it))) \simeq a(i(l_7(it)))$ occurs with the following instance:

$$\begin{aligned}\forall x_{\mathbb{I}}.\Big((Dense_{l_5,i} \wedge i(l_5(\mathtt{0})) \leq x < i(l_5(nl_5))) \rightarrow \\ \exists it_{\mathbb{N}}.(it < nl_5 \wedge\ i(l_5(it)) \simeq x \wedge \\ i(l_5(\mathtt{suc}(it))) \simeq i(l_5(it)) + 1)\Big).\end{aligned}$$

The reasoning here is not as straightforward as in Example 6. One would assume that the intermediate value theorem is applied to j since j is used as the access variable for array b. However, the automated proof uses this theorem on the access variable to a, namely i to determine the iteration and derives the update to b via the semantics of the conditional within the loop.

(B2) Iteration Injectivity Trace Lemma

Let w be a while-statement and let v be a mutable program variable. The *iteration injectivity trace lemma of w and v* is

$$\forall it^1_{\mathbb{N}}, it^2_{\mathbb{N}}.\Big((StrDense_{w,v} \wedge it^1 < it^2 \leq nl_{\mathtt{w}}) \\ \rightarrow v(tp_{\mathtt{w}}(it^1)) \not\approx v(tp_{\mathtt{w}}(it^2))\Big) \tag{B2}$$

The trace lemma (B2) states that a strongly-dense variable visits each array-position at most once. As a consequence, if each array position is visited only once in a loop, we know that its value has not changed after the first visit, and in particular the value at the end of the loop is the value after the first visit.

Example 8. For the property of Figure 3.1 we instantiate (B2) for j:

$$\forall it^1_{\mathbb{N}}, it^2_{\mathbb{N}}.\Big((StrDense_{l_6,j} \wedge it^1 < it^2 \leq nl_6) \\ \rightarrow j(l_6(it^1)) \not\approx j(l_6(it^2))\Big)$$

Trace lemma (B2) is necessary in Figure 3.1 to apply the value evolution trace lemma (A1) for b, as we need to make sure we will never reach the same position of j twice.

Example 9. Similarly, we apply trace lemma (B2) to j in Figure 3.2. Since j is used to access the mutable array variable b, we have to make sure that once a value is updated at b[j], we do not update the same position at a later timepoint. Hence we have

$$\forall it^1_{\mathbb{N}}, it^2_{\mathbb{N}}.\Big((Dense_{l_5,j} \wedge j(l_5(\mathtt{suc}(it^1)) = j(l_5(it^1)) + 1 \\ \wedge it^1 < it^2 \leq nl_5) \\ \rightarrow j(l_5(it^1)) \not\approx j(l_5(it^2))\Big).$$

In combination with the value evolution trace lemma (A1), this enables the prover to infer that once a value of b at some position is changed it will not be revisited or changed at a later time point at this particular position concluding the proof of the property in Figure 3.2. Note the difference in the proof compared to Example 8: the conditional in Figure 3.2 changes the nature of the proof. While the intermediate value theorem and the iteration injectivity lemmas are applied to the same program variable in Figure 3.1, namely j, the same is not sufficient to prove the safety property in Figure 3.2. Thus, even seemingly simple conditionals can change the need for trace lemmas throughout proof search.

3.3 Trace Lemma Correctness

In this section we prove soundness of trace lemmas (A1) and (B1)-(B2). We apply bounded induction (*B-Ind*) to establish the correctness of the above lemmas.

(Soundness of Value Evolution (A1)*).* Let bl and br be arbitrary but fixed and assume that the premise of the outermost implication of (A1) holds. That is,

$$\begin{aligned}\forall it_{\mathbb{N}}.((bl \leq it < br \wedge v(tp_{\mathrm{w}}(bl)) \circ v(tp_{\mathrm{w}}(it)))\\ \rightarrow v(tp_{\mathrm{w}}(bl)) \circ v(tp_{\mathrm{w}}(\mathtt{suc}(it))))\end{aligned} \tag{3.1}$$

We use the induction axiom scheme (*B-Ind*) and consider its instance with $P(it) := v(tp_{\mathrm{w}}(bl)) \circ v(tp_{\mathrm{w}}(it))$, yielding the following instance of (*B-Ind*):

$$\Big(v(tp_{\mathrm{w}}(bl)) \circ v(tp_{\mathrm{w}}(it)) \quad \wedge \tag{3.2a}$$

$$\forall it_{\mathbb{N}}.((bl \leq it < br \wedge v(tp_{\mathrm{w}}(bl)) \circ v(tp_{\mathrm{w}}(it))) \tag{3.2b}$$

$$\rightarrow v(tp_{\mathrm{w}}(bl)) \circ v(tp_{\mathrm{w}}(\mathtt{suc}(it))))\Big)$$

$$\rightarrow \forall it_{\mathbb{N}}.\Big(bl \leq it \leq br \rightarrow v(tp_{\mathrm{w}}(bl)) \circ v(tp_{\mathrm{w}}(it))\Big) \tag{3.2c}$$

Note that the base case property (3.2a) holds since $\circ$ is reflexive. Further, the inductive case (3.2b) holds since it is identical to the assumption (3.1). We thus derive property (3.2c), that is $\forall it_{\mathbb{N}}.\Big(bl \leq it \leq br \rightarrow v(tp_{\mathrm{w}}(bl)) \circ v(tp_{\mathrm{w}}(it))\Big)$. In particular, by instantiating it in the conclusion 3.2c to br yields $bl \leq br \leq br \rightarrow v(tp_{\mathrm{w}}(bl)) \circ v(tp_{\mathrm{w}}(br))$. By reflexivity of $\leq$ we conclude $bl \leq br \rightarrow v(tp_{\mathrm{w}}(bl)) \circ v(tp_{\mathrm{w}}(br))$, proving thus trace lemma (A1).

□

(Soundness of Intermediate Value (B1)*).* We prove the following equivalent contrapositive formula obtained from the intermediate value trace lemma (B1) by modus tollens.

$$\begin{aligned}\forall x_{\mathbb{I}}.\Big(\Big(&Dense_{w,v} \wedge v(tp_{\mathrm{w}}(\mathtt{0})) \leq x \wedge\\ &\forall it_{\mathbb{N}}.((it < nl_{\mathrm{w}} \wedge v(tp_{\mathrm{w}}(\mathtt{suc}(it))) \simeq v(tp_{\mathrm{w}}(it)) + 1)\\ &\rightarrow v(tp_{\mathrm{w}}(it)) \not\simeq x)\Big)\\ &\rightarrow v(tp_{\mathrm{w}}(nl_{\mathrm{w}})) \leq x\Big)\end{aligned} \tag{3.3}$$

The proof proceeds by deriving the conclusion of formula (3.3) from the premises of formula (3.3).
Consider the instance of the induction axiom scheme with

$$\text{Base case: } v(tp_{\mathrm{w}}(\mathtt{0})) \leq x \tag{3.4a}$$

$$\text{Inductive case: } \forall it_{\mathbb{N}}.\Big((\mathtt{0} \leq it < nl_{\mathrm{w}} \wedge v(tp_{\mathrm{w}}(it)) \leq x) \tag{3.4b}$$

$$\rightarrow v(tp_{\mathrm{w}}(\mathtt{suc}(it))) \leq x\Big)$$

$$\text{Conclusion: } \forall it_{\mathbb{N}}.\Big(\mathtt{0} \leq it \leq nl_{\mathrm{w}} \rightarrow v(tp_{\mathrm{w}}(it)) \leq x\Big), \tag{3.4c}$$

obtained from the bounded induction axiom scheme (*B-Ind*) with $P(it) := v(tp_{\mathrm{w}}(it)) \leq x$ for bounds $\mathtt{0}$ to nl_{w}.
The base case (3.4a) holds by assumption, as it is the second premise of (3.3). For the inductive case (3.4b), assume $\mathtt{0} \leq it < nl_{\mathrm{w}}$ and $v(tp_{\mathrm{w}}(it)) \leq x$. By density of v, we obtain two cases:

(1) Either we have $v(tp_{\mathrm{w}}(\mathtt{suc}(it))) = v(tp_{\mathrm{w}}(it))$. By assumption $v(tp_{\mathrm{w}}(it)) \leq x$ holds, hence we obtain $v(tp_{\mathrm{w}}(\mathtt{suc}(it))) \leq x$.
(2) Or we have $v(tp_{\mathrm{w}}(\mathtt{suc}(it))) = v(tp_{\mathrm{w}}(it)) + 1$. From (3.4b) we have $it < nl_{\mathrm{w}}$. We can thus apply the third premise of formula (3.3), and obtain $v(tp_{\mathrm{w}}(it)) \not\approx x$. Combined with our assumption $v(tp_{\mathrm{w}}(it)) \leq x$ and the totality-axiom of $<$ for integers, we have $v(tp_{\mathrm{w}}(it)) < x$. By integer theory, we thus have $v(tp_{\mathrm{w}}(\mathtt{suc}(it))) < x + 1$ and finally derive $v(tp_{\mathrm{w}}(\mathtt{suc}(it))) \leq x$.

This concludes the proof of the inductive case (3.4b). Thus, the conclusion (3.4c) holds. Since the theory axiom $\forall it_{\mathbb{N}}.\ \mathtt{0} \leq it$ holds, formula (3.4c) implies the conclusion of formula (3.3), which concludes the proof. □

(Soundness of Iteration Injectivity (B2)*).* For arbitrary but fixed iterations $it^1_{\mathbb{N}}$ and $it^2_{\mathbb{N}}$, assume that the premises of the lemma hold. Now consider the instance of the induction axiom scheme with

$$\text{Base case: } v(tp_{\mathrm{w}}(it^1)) < v(tp_{\mathrm{w}}(\mathtt{suc}(it^1))) \tag{3.5a}$$

$$\text{Inductive case: } \forall it_{\mathbb{N}}.\Big((\mathtt{suc}(it^1) \leq it < nl_{\mathrm{w}} \wedge v(tp_{\mathrm{w}}(it^1)) < v(tp_{\mathrm{w}}(it))) \to v(tp_{\mathrm{w}}(it^1)) < v(tp_{\mathrm{w}}(\mathtt{suc}(it)))\Big) \tag{3.5b}$$

$$\text{Conclusion: } \forall it_{\mathbb{N}}.\Big(\mathtt{suc}(it^1) \leq it \leq nl_{\mathrm{w}} \to v(tp_{\mathrm{w}}(it^1)) < v(tp_{\mathrm{w}}(it))\Big), \tag{3.5c}$$

obtained from the bounded induction axiom scheme (*B-Ind*) with $P(it) := v(tp_{\mathrm{w}}(it^1)) < v(tp_{\mathrm{w}}(it))$, by instantiating *bl* and *br* to $\mathtt{suc}(it^1)$, respectively nl_{w}.
The base case (3.5a) holds since by integer theory we have $\forall x_{\mathbb{I}}.\ x < x + 1$ and by assumption *StrDense*$_{w,v}$ hence $v(tp_{\mathrm{w}}(\mathtt{suc}(it^1))) = v(tp_{\mathrm{w}}(it^1)) + 1$ holds.
For the inductive case, we assume for arbitrary but fixed it that $v(tp_{\mathrm{w}}(it^1)) < v(tp_{\mathrm{w}}(it))$ holds. Combined with *StrDense*$_{w,v}$ and $\forall x_{\mathbb{I}}.(x < y \to x < y + 1)$ this yields $v(tp_{\mathrm{w}}(it^1)) < v(tp_{\mathrm{w}}(\mathtt{suc}(it)))$, so (3.5b) holds. Since both premises (3.5a) and (3.5b) hold, also the conclusion (3.5c) holds. Next, $it^1 < it^2$ implies $\mathtt{suc}(it^1) \leq it^2$ (using the monotonicity of $\mathtt{suc}$). We therefore have $\mathtt{suc}(it^1) \leq it^2 < nl_{\mathrm{w}}$, so we are able to instantiate the conclusion(3.5c) to obtain $v(tp_{\mathrm{w}}(it^1)) < v(tp_{\mathrm{w}}(it^2))$. Finally, we use the arithmetic property $\forall x_{\mathbb{I}}, y_{\mathbb{I}}.(x < y \to x \not\approx y)$ to conclude $v(tp_{\mathrm{w}}(it^1)) \not\approx v(tp_{\mathrm{w}}(it^2))$. □

Based on the soundness of our trace lemmas, we conclude the next result.

Theorem 3 (Trace Lemmas and Induction)**.** *Let p be a program. Let L be a trace lemma for some while-statement w of p and some variable v of p. Then L is a consequence of the bounded induction scheme (B-Ind) and of the axiomatic semantics of $[\![p]\!]$ in trace logic $\mathcal{L}$.*

Our work is implemented in the RAPID verification framework and relies on the VAMPIRE theorem prover. For implementation details and the experimental evaluation, we refer to Chapter 7 and Chapter 8 respectively.

3.4 Related Work

Our work is closely related to recent efforts in using first-order theorem provers for proving software properties [KV09, GKR18]. While [GKR18] captures program semantics in the first-order language of extended expressions over loop iterations, in our work we further generalize the semantics of program locations and consider program expressions over loop iterations and arbitrary timepoints. We introduce and prove trace lemmas to automate inductive reasoning based on bounded induction over loop iterations. Our generalizations in trace logic proved to be necessary to automate the verification of properties with arbitrary quantification, which could not be effectively achieved in [GKR18]. Our work is not restricted to reasoning about single loops as in [GKR18].

A variation of our approach has already been successfully applied to relational verification in [BEG+19] to prove 2-safety properties of programs such as non-interference and sensitivity. Recent developments in first-order theorem proving allowed us to generalize these ideas to a wider setting of provable properties. Compared to [BEG+19], we ensure soundness of our trace lemmas for safety verification.

In comparison to verification approaches based on program transformations [KFG20, CGU20, YFG19], we do not require user-provided functions to transform program states to smaller-sized states [ISIRS20], nor are we restricted to universal properties generated by symbolic execution [CGU20]. Rather, we use only three trace lemmas that we prove sound and automate the verification of first-order properties, possibly with alternations of quantifiers.

The works [DDA10, CCL11] consider expressive abstract domains and limit the generation of universal invariants to these domains, while supporting potentially more generic program grammars than our $\mathcal{W}$ language. Our work, however, can verify universal and/or existential first-order properties with theories, which is not the case in [KFG20, CGU20, DDA10, CCL11]. Verifying universal loop properties with arrays by implicitly finding invariants is addressed in [GSV18, FPMG19, KBGM15, FKB17, FB18, MTK20], and by using constraint Horn clause reasoning within property-driven reachability analysis in [HB12, CG12].

Another line of research proposes abstraction and lazy interpolation [ABG+12, ACC+20], as well as recurrence solving with SMT-based reasoning [RL18]. Synthesis-based approaches, such as [FPMG19], are shown to be successful when it comes to inferring universally quantified invariants and proving program correctness from these invariants.

Synthesis-based term enumeration is used also in [YFG19] in combination with user-provided invariant templates. Compared to these works, we do not consider programs only as a sequence of states, but model program values as functions of loop iterations and timepoints, allowing thus to express program semantics over sequences of sequences of states. Further, we use trace logic reasoning to synthesize bounds on loop iterations and infer first-order loop invariants as logical consequences of our trace lemmas and program semantics in trace logic.

CHAPTER 4

Lemmaless Inductive Reasoning

This chapter extends work published in

> Ahmed Bhayat, Pamina Georgiou, Clemens Eisenhofer, Laura Kovács, and Giles Reger. Lemmaless Induction in Trace Logic. In *International Conference on Intelligent Computer Mathematics (CICM 2022)* , pages 191-208. Springer, 2022.

At a high level, our verification framework based on trace lemma reasoning [GGK20a] works by translating a program into *trace logic*, adding a number of ad hoc trace lemmas, asserting a desired property, and then running an automated theorem prover on the result. The effectiveness of this approach depends not only on the underlying trace lemmas but also on the search space - adding trace lemmas automatically always results in unnecessary instantiations that are not helpful for proof search but rather increase the search space. However, static analysis is not enough to prune this search space efficiently. This chapter thus focuses on building induction support into the underlying theorem prover Vampire to reduce reliance on trace lemma reasoning as introduced in Chapter 3. Trace logic is an instance of first-order logic with theories, such that the program semantics of imperative programs with loops, branching, integers, and arrays can be directly encoded in a formal manner. A key feature of this encoding is tracking program executions by quantifying over execution *timepoints* (rather than only over single states), which are themselves be parameterised by *loop iterations*. In principle, we can check whether a translated program entails the desired property in trace logic using an automated theorem prover for first-order logic. In our case, we make use of the saturation-based theorem prover Vampire which implements the superposition calculus [BG01]. However, a straightforward use of theorem proving often fails in establishing validity of program properties in trace logic, as the proof requires some specific induction, in general not supported by superposition-based reasoning. Thus, automating inductive reasoning in trace logic remains a challenge in order to prove inductive program properties over timepoints of program execution.

In our prior work [GGK20a], we overcame this challenge by introducing trace lemmas (see Chapter 3.2) capturing common patterns of inductive loop properties over arrays and integers. Inductive loop reasoning in trace logic is then achieved by generating and adding trace lemma instances to the translated program. However, there are three significant limitations to using trace lemmas:

1. Trace lemmas capture inductive patterns/templates that need to be manually identified, as induction is not generally expressible in first-order logic. As such, they cannot be inferred by a first-order reasoner, implying that the effectiveness of trace logic reasoning depends on the expressiveness of manually supplied trace lemmas.
2. When instantiating trace lemmas with appropriate inductive program variables, a large number of inductive properties are generated, causing saturation-based proof search to diverge and fail to find program correctness proofs in reasonable time. This means that, even for small input programs, it is quite common that too many trace lemma instances are generated, decreasing the efficiency of first-order proving.
3. Certain necessary inductive properties, for instance to relate values of two or more program variables require too many automatically instantiated additions to the input problem, thus causing more harm than good. Such lemmas exceed the capabilities of trace lemma reasoning with first-order theorem provers as adding them for all adequate combinations explodes the search space and renders proof search divergent.

Recent advances in first-order theorem proving automating integer and structural induction [HKV21, HHK+20] have paved the way for exploring automated inductive reasoning for programs containing loops and unbounded data structures natively in the first-order reasoner. However, the main challenge remains to find reasonable terms to induct upon. In this chapter we address these limitations and challenges by drastically reducing the need for trace lemmas. We achieve this by introducing a couple of novel induction inferences specialized for the software verification setting in trace logic. That is, we include inductive inferences based on bounded induction over loop iterations directly in the underlying theorem prover such that they are performed during proof search. Firstly, *multi-clause goal induction* which applies induction in a goal oriented fashion as many safety program assertions are structurally close to useful loop invariants. Secondly, *array mapping induction* which covers certain cases where the required loop invariant does not stem from the goal but rather depends on the program semantics. Specifically, we make the following contributions:

Contribution 1. We introduce two new inference rules, *multi-clause goal* and *array mapping* induction, for *lemmaless induction* over loop iterations (Sections 4.3–4.4). The inference rules are compatible with any saturation-based inference system used for first-order theorem proving and work by carrying out induction on terms corresponding to final loop iterations.

Contribution 2. We implemented our approach in the first-order theorem prover Vampire [KV13]. Further, we extended the Rapid framework [GGK20a] to support inductive reasoning in the automated backend. We describe such implementation details in Chap-

```
 1  func main() {
 2    const Int[] a;
 3    const Int[] b;
 4    Int[] c;
 5    const Int length;
 6    Int i = 0;
 7
 8    while (i < length) {
 9      c[2*i] = a[i]
10      c[(2*i) + 1] = b[i]
11      i = i + 1;
12    }
13  }
14  assert(∀pos_𝕀.∃l_𝕀.((0 ≤ pos < (2 × length)) → c(end, pos) = a(l) ∨ c(end, pos) = b(l)))
```

Figure 4.1: Program copying elements from arrays `a` and `b` to even/odd positions in array `c`.

ter 7. We carry out an extensive evaluation of trace lemma versus *lemmaless* reasoning (see Chapter 8) and compare against state-of-the-art approaches SeaHorn [GSV18, GKKN15] and Vajra/Diffy [CGU20, CGU21].

4.1 Motivating Example

We motivate our work with the example program in Figure 4.1. The program iterates over two arrays `a` and `b` of arbitrary, but fixed length `length` and copies array elements into a new array `c`. Each even position in `c` contains an element of `a`, while each odd position an element of `b`. Our task is to prove the safety assertion at line 14: at the end of the program, every element in `c` is an element from `a` or `b`. Note that the length of array `c` is twice `length` after computation. This property involves (i) alternation of quantifiers and (ii) is expressed in the first-order theories of linear integer arithmetic and arrays. In the safety assertion, the program variable `length` is modeled as a logical constant of the same name of sort integer, whilst the constant arrays `a` and `b` are modeled as logical functions from integers to integers. The mutable array variable `c` is additionally equipped with a timepoint argument *end*, indicating that the assertion is referring to the value of the variable at the end of program execution.
Proving the correctness of this example program remains challenging for most state-of-the-art approaches, such as [GKKN15, FPMG19, CGU20, CGU21], mainly due to the complex quantified structure of our assertion. Moreover, it cannot be achieved in the trace lemma framework either, as existing trace lemmas do not relate the values of multiple program variables, notably equality over multiple array variables. In fact, to automatically prove the assertion, we need an inductive property/trace lemma formalizing that each element at an even position in `c` is an element of `a` or `b` at each valid loop iteration, thereby also restricting the bounds of the loop counter variable `i`. Naïvely adding such a

trace lemma would be highly inefficient as automated generation of verification conditions would introduce many instances that are not required for the proof. Similarly, undirected application of induction rules within the first-order prover would lead to an even more dramatic explosion of the search space. Some guidance on what terms to induct on is clearly necessary. To this end, we introduce two inductive multi-clausal inference rules, namely multi-clause goal induction (Section 4.3) and array-mapping induction (Section 4.4), that specialize the terms we induct upon. While these inference rules dramatically reduce the need for trace lemma based reasoning, we introduce one new trace lemma (Section 4.2) that is necessary to handle a small but significant part of the reasoning task that is not covered by built-in induction.

4.2 A Final Trace Lemma

We directly equip the first-order reasoner with multi-clause induction on natural numbers specifically designed for trace logic. This mechanism allows the prover to find the induction hypotheses automatically and relies on proving the base and step case respectively. Nonetheless, due to some limitations in our first-order prover, we are unable to completely do away with additional lemmas. Specifically, we need to nudge the prover to deduce that a loop counter variable will, at the end of loop execution, have the value of the expression it is compared against in the loop condition. We thus introduce the *Equal Lengths Trace Lemma* for increasing loop counter variables:

(C) Equal Lengths Trace Lemma

Let $\mathtt{w}$ be a while-statement, $C_{\mathtt{w}} := \mathtt{e} < \mathtt{e'}$ be the loop condition where $\mathtt{e'}$ is a program expression that remains constant during the execution of $\mathtt{w}$. The *equal lengths trace lemma of* $\mathtt{w}$, $\mathtt{e}$ *and* $\mathtt{e'}$ is defined as

$$\begin{aligned}(Dense_{w,e} \wedge [\![\mathtt{e}]\!](tp_{\mathtt{w}}(\mathtt{0})) \leq [\![\mathtt{e'}]\!](tp_{\mathtt{w}}(\mathtt{0}))) \rightarrow \\ [\![\mathtt{e}]\!](tp_{\mathtt{w}}(nl_{\mathtt{w}})) \simeq [\![\mathtt{e'}]\!](tp_{\mathtt{w}}(nl_{\mathtt{w}})).\end{aligned} \tag{C}$$

Trace lemma C states that a dense expression $\mathtt{e}$ smaller than or equal to some expression $\mathtt{e'}$ that does not change in the loop, will eventually, specifically in the last iteration, reach the same value as $\mathtt{e'}$. This follows from the fact that we assume termination of a loop, hence we assume the existence of a timepoint $nl_{\mathtt{w}}$ where the loop condition does not hold anymore. As a consequence, given that the loop condition held at the beginning of the execution, we can derive that the loop counter value immediately after the loop execution $[\![\mathtt{e}]\!](tp_{\mathtt{w}}(nl_{\mathtt{w}}))$ will necessarily equate to $[\![\mathtt{e'}]\!](tp_{\mathtt{w}}(\mathtt{0})) = [\![\mathtt{e'}]\!](tp_{\mathtt{w}}(nl_{\mathtt{w}}))$. In the special case where $\mathtt{e'}$ contains no mutable variables, the conclusion of the lemma can be simplified to $[\![\mathtt{e}]\!](tp_{\mathtt{w}}(nl_{\mathtt{w}})) = [\![\mathtt{e'}]\!]$. Note that a similar lemma can just as easily be added for dense but decreasing loop counters.

Example 10. For the program in Figure 4.1, we instantiate lemma C for variable `i` and obtain

$$(Dense_{l_8,i} \wedge i(l_8(0)) \leq length) \rightarrow i(l_8(nl_8)) \simeq length.$$

4.3 Multi-Clause Goal Induction for Lemmaless Induction

The main focus of our work is moving induction into the saturation prover with the aim of proving inductive program properties using only the trace logic progam semantics, thus reducing the need for extra trace lemmas. We achieve this by adding inference rules that apply bounded induction over loop iterations directly in the underlying theorem prover. To this end, we identify loop counter terms and leverage recent theorem proving efforts on *bounded (integer) induction* in saturation [HKV21, HHK+20]. However, as illustrated in the following, these recent efforts cannot be directly used in trace logic reasoning since we need to (i) adjust bounded induction for the setting of loop iterations, i.e. natural numbers, and (ii) generalize to multi-clause induction. We discuss these steps using Figure 4.1. Verifying the safety assertion of Figure 4.1 requires proving the trace logic formula:

$$\begin{aligned} &\forall pos_{\mathbb{I}}.\, \exists j_{\mathbb{I}}.\, (0 \leq pos < (2 \times length) \\ &\quad \rightarrow (c(end, pos) \simeq a(j) \vee c(end, pos) \simeq b(j)) \end{aligned} \tag{4.1}$$

For proving (4.1), it suffices to prove that the following, slightly modified statement is a loop invariant of Figure 4.1:

$$\begin{aligned} \forall it_{\mathbb{N}}.\, it < nl_8 \rightarrow \forall pos_{\mathbb{I}}.\, \exists j_{\mathbb{I}}.\, (0 \leq pos < (2 \times i(tp_{\mathtt{w}}(it)))) \\ \rightarrow (c(l_8(it), pos) \simeq a(j) \vee c(l_8(it), pos) \simeq b(j)) \end{aligned} \tag{4.2}$$

where l_8 refers to the time point of the loop statement in Figure 4.1. As part of the program semantics in trace logic, we have formula (4.3) which links the value of c at the end of the loop (iteration nl_8) to its value at the end of the program. Moreover, using the trace lemma C, we also derive formula (4.4) in trace logic:

$$\forall pos_{\mathbb{I}}.c(l_8(nl_8), pos) \simeq c(end, pos) \tag{4.3}$$

$$i(l_8(nl_{\mathtt{w}})) \simeq length \tag{4.4}$$

It is tempting to think that in the presence of these clauses (4.3)–(4.4), a saturation-based prover would rewrite the negated conjecture (4.1) to

$$\begin{aligned} \neg(\forall pos_{\mathbb{I}}.\, \exists j_{\mathbb{I}}.\, (0 \leq pos < (2 \times i(l_8(nl_8)))) \\ \rightarrow (c(l_8(nl_8), pos) \simeq a(j) \vee c(l_8(nl_8), pos) \simeq b(j))) \end{aligned}$$

from which a bounded natural number induction inference (similar to the $\mathtt{IntInd}_<$ rule of [HKV21]) would quickly introduce an induction hypothesis with (4.2) as the conclusion,

by induction over nl_8. However, this is not the case, as most saturation provers work by first *clausifying* their input. The negated conjecture (4.1) would not remain a single formula, but be split into the following four clauses where sk is a Skolem symbol:

$$\begin{array}{cc} a(x) \not\simeq c(end, sk) & b(x) \not\simeq c(end, sk) \\ \neg(sk \leq 0) & sk \leq 2 \times length \end{array}$$

These clauses can be rewritten using (4.3)–(4.4). For example, the first clause can be rewritten to $a(x) \not\simeq c(l_8(nl_8, sk))$. However, attempting to prove the negation of any of the rewritten clauses individually via induction would merely result in the addition of useless induction formulas to the search space. For example, attempting to prove $\forall it_{\mathbb{N}}.\, it < nl_8 \rightarrow (\exists x_{\mathbb{I}}.\, a(x) \simeq c(l_8(it), sk))$, is pointless as it is clearly false. *The solution we propose in this work is to use multi-clause induction*, whereby we attempt to prove the negation of the conjunction of multiple clauses via a single induction inference. For our example in Figure 4.1, we can use the following rewritten versions of clauses from the negated conjecture $a(x) \not\simeq c(l_8(nl_8, sk))$, $b(x) \not\simeq c(l_8(nl_8, sk))$, and $sk \leq 2 \times i(l_8(nl_8))$, with induction term nl_8, to obtain the following multi-clause induction formula:

$$\begin{array}{l} \neg\Big(\quad \forall x_{\mathbb{I}}.\, a(x) \not\simeq c(i(l_8(0)), sk) \\ \quad \wedge\, \forall x_{\mathbb{I}}.\, b(x) \not\simeq c(i(l_8(0), sk)) \\ \quad \wedge\, sk \leq 2 \times i(l_8(0))\Big) \\ \wedge\, \mathit{StepCase} \end{array} \rightarrow \begin{array}{l} \forall it_{\mathbb{N}}.\, it < nl_8 \rightarrow \\ \quad \neg\Big(\quad \forall x_{\mathbb{I}}.\, a(x) \not\simeq c(i(l_8(it), sk) \\ \quad\; \wedge \forall x_{\mathbb{I}}.\, b(x) \not\simeq c(i(l_8(it), sk) \\ \quad\; \wedge\, sk \leq 2 \times i(l_8(it))\Big) \end{array} \tag{4.5}$$

where *StepCase* is the formula:

$$\begin{array}{l} \forall it_{\mathbb{N}}.\, it < nl_8 \wedge \\ \neg\Big(\quad \forall x_{\mathbb{I}}.\, a(x) \not\simeq c(i(tp_{\mathtt{w}}(it)), sk) \\ \quad \wedge\, \forall x_{\mathbb{I}}.\, b(x) \not\simeq c(i(tp_{\mathtt{w}}(it)), sk) \\ \quad \wedge\, sk \leq i(tp_{\mathtt{w}}(y))\Big) \end{array} \rightarrow \begin{array}{l} \neg\Big(\quad \forall x_{\mathbb{I}}.\, a(x) \not\simeq c(i(tp_{\mathtt{w}}(\mathtt{suc}(it)), sk) \\ \quad \wedge \forall x_{\mathbb{I}}.\, b(x) \not\simeq c(i(tp_{\mathtt{w}}(\mathtt{suc}(it)), sk) \\ \quad \wedge\, sk \leq 2 \times i(tp_{\mathtt{w}}(\mathtt{suc}(it)))\Big) \end{array}$$

Using the induction formula (4.5), a contradiction can then easily be derived, establishing validity of (4.1). In what follows, we formalize the multi-clause induction principle we used above. To this end, we introduce a generic inference rule, called *multi-clause goal induction* and denoted as `MCGLoopInd`.

$$\frac{C_1[nl_{\mathtt{w}}] \qquad C_2[nl_{\mathtt{w}}] \qquad \ldots \qquad C_n[nl_{\mathtt{w}}]}{\mathrm{CNF}\left(\begin{array}{l} \neg(C_1[0] \wedge C_2[0] \wedge \ldots \wedge C_n[0]) \wedge \\ \forall it_{\mathbb{N}}.\, \left(\begin{array}{l} ((it < nl_{\mathtt{w}}) \wedge \neg(C_1[it] \wedge C_2[it] \wedge \ldots \wedge C_n[it])) \rightarrow \\ \neg(C_1[\mathtt{suc}(it)] \wedge C_2[\mathtt{suc}(it)] \wedge \ldots \wedge C_n[\mathtt{suc}(it)])) \end{array}\right) \\ \rightarrow (\forall it_{\mathbb{N}}.\, (it < nl_{\mathtt{w}}) \rightarrow \neg(C_1[it] \wedge C_2[it] \wedge \ldots \wedge C_n[it])) \end{array}\right)}$$

For performance reasons, we mandate that the premises $C_1 \ldots C_n$ be derived from trace logic formulas expressing safety assertions and not from formulas encoding the program semantics. The `MCGLoopInd` rule is formalized only as an induction inference over last loop iteration symbols. While restricting to $nl_{\mathtt{w}}$ terms is of purely heuristic nature, our experiments justify the necessity and usefulness of this condition (Section 8.1).

```
 1  func main(){
 2    const Int alength;
 3    Int[] a;
 4    Int i = 0;
 5    const Int n;
 6
 7    while(i < alength){
 8      a[i] = a[i] + n;
 9      i = i + 1;
10    }
11
12    Int j = 0;
13    while(j < alength){
14      a[j] = a[j] - n;
15      j = j + 1;
16    }
17  }
18  assert(∀pos_𝕀.((0 ≤ pos < alength) → a(end, pos) = a(start, pos)))
19
```

Figure 4.2: Program that adds and subtracts `n` to every element of array `a`.

4.4 Array Mapping Induction for Lemmaless Induction

Multi-clause goal induction neatly captures goal-oriented application of induction. Nevertheless, there are verification challenges where `MCGLoopInd` fails to prove inductive loop properties. This is particularly the case for benchmarks containing multiple loops, such as in Figure 4.2. We first discuss the limitations of `MCGLoopInd` using Figure 4.2, after which we present our solution, the *array mapping induction* inference.

Let $\mathtt{w}_1$ be the first loop statement of Figure 4.2 and $\mathtt{w}_2$ be the second loop. Let l_7 and l_{13} denote the timepoints of the first and second loop with nl_7 and nl_{13} being their last iteration symbols respectively. Using `MCGLoopInd`, we would attempt to prove

$$\begin{aligned}&\forall it_{\mathbb{N}}.\, it < nl_{13} \rightarrow \\ &\quad \forall pos_{\mathbb{I}}.\, (0 \leq pos < j(l_{13}(it))) \rightarrow (a(l_{13}(it), pos) \simeq a(start, pos)\end{aligned} \tag{4.6}$$

However, formula (4.6) is not a useful invariant for proving the assertion. Since the prior loop at location l_7 changes the original contents of array `a`, we cannot derive the above induction axiom. Rather, for $\mathtt{w}_2$ we need a loop invariant similar to

$$\begin{aligned}\forall it_{\mathbb{N}}.\, it < nl_{13} \rightarrow \forall pos_{\mathbb{I}}.\, (0 \leq pos < j(l_{13}(it))) \\ \rightarrow (a(l_{13}(it), pos) \simeq a(l_{13}(0), pos) - n\end{aligned} \tag{4.7}$$

and an equivalent invariant for loop $\mathtt{w}_1$. The loop invariant (4.7) is however not linked to the safety assertion of Figure 4.2, and thus multi-clause goal induction is unable to infer and prove with it. To aid with the verification of benchmarks such as Figure 4.2, we introduce another induction inference which we call *array mapping induction*. In this

case, we trigger induction not on clauses and terms coming from the goal, but on clauses and terms appearing in the program semantics.
The *array mapping induction* inference rule, denoted as `AMLoopInd` is given below. Essentially, `AMLoopInd` involves analyzing a clause set to heuristically devise a suitable loop invariant. Guessing a candidate loop invariant is a difficult problem. The `AMLoopInd` inference is triggered if clauses of the shapes of C_1 and C_2 defined below are present in the clause set. Intuitively, C_1 states that i increases by m in each iteration of the loop. Clause C_2 can be read as saying that on each round of some loop $\mathtt{w}$, some array a at position i is set to some function F of its previous value at that position.
Together the two clauses suggest that the loop is mapping the function F to each mth location of the array starting from the array cell located at $i(tp_{\mathtt{w}}(0))$. This is precisely what the induction formula attempts to prove. Note that for ease of notation, we present the inference for the case where the indexing variable is *increasing*. It is straightforward to generalise to the decreasing case. The `AMLoopInd` rule is[1]

$$\frac{\begin{array}{c} C_1 = i(tp_{\mathtt{w}}(\mathtt{suc}(x))) \simeq i(tp_{\mathtt{w}}(x)) + m \ \vee \ \neg(x < nl_{\mathtt{w}}) \\ C_2 = a(tp_{\mathtt{w}}(\mathtt{suc}(x)), i(tp_{\mathtt{w}}(x))) \simeq F[a(tp_{\mathtt{w}}(x), i(tp_{\mathtt{w}}(x)))] \ \vee \ \neg(x < nl_{\mathtt{w}}) \end{array}}{\mathrm{CNF}(\mathit{StepCase} \rightarrow \mathit{Conclusion})}$$

where $\mathtt{w}$ is some loop and F an arbitrary non-empty context. Let i_0 be an abbreviation for $i(tp_{\mathtt{w}}(0))$. Then:

$$\begin{aligned} \mathit{StepCase}: \quad & \forall it_{\mathbb{N}}.\, (\forall y_{\mathbb{I}}.\, it < nl_{\mathtt{w}} \wedge \\ & \quad y < i(tp_{\mathtt{w}}(it)) - i_0 \wedge y \geq 0 \wedge y \bmod m = 0 \\ & \quad\quad \rightarrow a(tp_{\mathtt{w}}(it), i_0 + y) \simeq F[a(tp_{\mathtt{w}}(0), i_0 + y)]) \rightarrow \\ & (\forall y_{\mathbb{I}}.\, y < i(tp_{\mathtt{w}}(\mathtt{suc}(it))) - i_0 \wedge y \geq 0 \wedge y \bmod m = 0 \\ & \quad\quad \rightarrow a(tp_{\mathtt{w}}(\mathtt{suc}(it)), i_0 + y) \simeq F[a(tp_{\mathtt{w}}(0), i_0 + y)]) \\ \mathit{Conclusion}: \ & \forall x_{\mathbb{I}}.\, x < i(tp_{\mathtt{w}}(nl_{\mathtt{w}})) - i_0 \wedge x \geq 0 \wedge x \bmod m = 0 \\ & \quad\quad \rightarrow a(tp_{\mathtt{w}}(nl_{\mathtt{w}}), i_0 + x) \simeq F[a(tp_{\mathtt{w}}(0), i_0 + x)] \end{aligned}$$

To prove *StepCase*, it is necessary to be able to reason that positions in the array a remain unchanged until visited by the indexing variable. This can be achieved via the addition of another induction to the conclusion of the inference. Our approach is implemented as an extension of the Rapid framework, using the first-order theorem prover Vampire described in Chapter 7, Section 7.2.2. The `AMLoopInd` inference is thus sufficient to prove the assertion of Figure 4.2. While `AMLoopInd` is a limited approach for guessing inductive loop invariants, we believe it can be extended towards further, more generic methods to guess invariants, as discussed in Chapter 9. We conclude this section by noting that our induction rules are sound, based on trace logic semantics. Since both rules merely add instances of the bounded induction schema for natural numbers (*B-Ind*) to the search space, soundness is trivial and we do not provide a proof.

[1] In the conclusion we ignore the base case of the induction formula as it is trivially true.

Theorem 4 (Soundness of Lemmaless Induction)**.** *The inference rules* `MCGLoopInd` *and* `AMLoopInd` *are sound.*

4.5 Related Work

Most of recent research in verifying inductive properties of array-manipulating programs focuses on quantified invariant generation is mostly restricted to proving universally quantified program properties. The works [GSV18, FPMG19] generate universally quantified inductive invariants by iteratively inferring and strengthening candidate invariants. These methods use SMT solving and as such are restricted to first-order theories with a finite model property. Similar logical restrictions also apply to [RL18], where linear recurrence solving is used in combination with array-specific proof tactics to prove quantified program properties. A related approach is described in [CGU21], where relational invariants instead of recurrence equations are used to handle universal and quantifier-free inductive properties. Unlike these works, our work is not limited to universal invariants but can both infer and prove inductive program properties with alternations of quantifiers.
With the use of extended expressions and induction schemata, our work shares some similarity with template-based approaches [SG09, LRCR13, KM16]. These works [SG09, LRCR13, KM16] infer and prove universal inductive properties based on Craig interpolation, formula slicing and/or SMT generalizations over quantifier-free formulas. Unlike these works, we do not require any assumptions on the syntactic shape of the first-order invariants. Moreover, our invariants are not restricted to the shape of our induction schemata. Rather, we treat inductive (invariant) inferences as additional rules of first-order theorem provers, maintaining thus the efficient handling of arbitrary first-order quantifiers. Our framework can be used in arbitrary first-order theories, even with theories that have no interpolation property and/or a finite axiomatization, as exemplified by our experimental results using inductive reasoning over arrays and integers.
Inductive theorem provers (ITP), such as ACL2 [KM97] and HipSpec [CJRS13], implement powerful induction schemata and heuristics. However these provers, to the best of our knowledge, automate inductive reasoning for only universally quantified inductive formulas using a goal/subgoal architecture, for which user-guidance is needed to split conjectures into subgoals. In contrast, our work can prove formulas of full first-order logic by integrating and fully automating induction in saturation-based proof search. By combining induction with saturation, we allow these techniques to interleave and complement each other, something that pure induction provers cannot do. Unlike tools such as Dafny [Lei10], our approach is fully automated requiring no user annotations.
Another technique used to heuristically guide induction in ITP is *rippling* [BSVH$^+$93]. Rippling deductively steers the goal towards the induction hypothesis through applications of rewriting, thereby reducing arbitrary rule applications of ITP that will likely not result in a proof. While, at first glance, this may sound similar to our multi-clause goal induction inference rule that applies to clauses derived from the safety assertion, hence the goal, our approach is fundamentally different to ITP. Our heuristics to guide induction are to some extent built-in in the inference rules by restricting their applications only to certain

clauses of the search space. Both our induction inference rules add new formulas to the search space and can thus replace the tactics of [BSVH+93] by integrating induction directly in superposition-based reasoning.
First-order theorem proving has previously been used to derive invariants with alternations of quantifiers in our previous work [GGK20a]. Our current work generalizes the inductive capabilities of [GGK20a] by reducing the expert knowledge of [GGK20a] in introducing inductive lemmas to guide the process of proving inductive properties.

CHAPTER 5

Extracting Invariants with Trace Lemma Reasoning

Partial results of this chapter are published in

> Pamina Georgiou, Bernhard Gleiss, Ahmed Bhayat, Michael Rawson, Laura Kovács, and Giles Reger. The Rapid Software Verification Framework. In *Proceedings of the 22th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2022)*, pages 255-260. IEEE, 2022.

In the previous two chapters, we formalized and proved safety assertions in trace logic for programs containing integers and arrays with various forms of automated inductive reasoning. However, synthesizing loop invariants is just as significant of a challenge as establishing automated proofs.
Invariants can be a useful tool for continuous software verification: annotating already verified code simplifies the task of re-establishing the validity of functional specifications to potential changes in the code. Extracting loop invariants also allows integration with other methods of verification. They can be reused for *compositionally* automated proofs by integrating them in various other tools such as Dafny [Lei10] or Why3 [FP13].Thus many state-of-the-art verification tools that handle array theory are based on invariant synthesis [FPMG19, PSM16, KBI+17].
When it comes to reasoning in the combination of recursive data structures such as unbounded arrays and linear arithmetic, such invariants usually contain quantification. In the realm of first-order theorem provers, the *symbol elimination* method [KV09] has been exploited to generate quantified invariants [KV09, GKR18, HKV11]. We adopt and revise this method for trace logic $\mathcal{L}$.

5.1 Extended Expressions and Symbol Elimination

Extended Expressions. Previous works such as [KV09] were based on program/loop semantics with so-called *extended expressions*: for any variable v appearing in a loop L, define an expression $v^{(i)}$ designating the value of v in the loop state σ_i, that is at some iteration i. Let σ_0 be an initial state for the computation of a loop. A formula Φ potentially containing extended expressions, consequently, is valid for L if it is true for any computation of L, that is $\forall i.\ i \geq 0 \rightarrow \Phi(i)$. Using extended expressions (and disregarding loop conditions) allowed to generate quite general inductive properties about loops and their program variables through static analysis. An example of such auxiliary loop properties are monotonicity properties: e.g. given a scalar program variable v that is dense and strictly increasing, we can conclude that $\forall i.\ v^{(i)} = v^{(0)} + i$. Note that extended expressions, in contrast to expressions in trace logic $\mathcal{L}$, define values merely over iterations, not timepoints. A loop is, therefore, semantically regarded as a sequence of states such that each iteration represents one such state. In contrast, our semantics in $\mathcal{L}$ define multiple states for each iteration, hence are more fine-grained.
However, arrays were handled in this setting as well: rather than using full-fledged program semantics such as in trace logic, the authors introduced so-called *update predicates* defining array updates at some position p at loop iteration i by some value v. These predicates allowed on the one hand to capture program behavior for loops that update arrays, and on the other, to determine inductive properties with extended expressions over such updates. For example, if an array `a` is updated at some position p at iteration i but is not updated in any further iteration, then `a[p]` will have the value that was assigned at iteration i at the end of the computation of the loop. That is, $upd_{\mathtt{a}}(i, p, v) \wedge \forall j.\ j > i \rightarrow a^{(n)}[p] = v$ for some value v and the final iteration n. Note that our value evolution theorem (A1) is a generalization of this property, allowing us to deduce the equivalent in $\mathcal{L}$ based on bounded induction.
Using an automated theorem prover on such a formalism produces many consequences. However, these consequences might contain many of the auxiliary symbols introduced by the use of extended expressions, respectively trace logic $\mathcal{L}$. Consequently, to obtain loop invariants in first-order logic, these auxiliary symbols need to be eliminated while finding proper consequences of the program/loop semantics.

Symbol Elimination. Symbol elimination with automated theorem proving was first introduced in [KV09] and is based on *symbol-eliminating inferences*. The *symbol elimination* approach defined some set of program symbols undesirable, and only reports consequences that have *eliminated* such symbols from their predecessors. Essentially, the mechanism deals with ridding the consequences of clauses that contain extended expressions or are purely theory-based conclusions. To be able to express interesting properties without the use of extended expressions, it is necessary to define new symbols for program variables, so-called *target symbols*. The idea is the following: for any program variable v, integer or array, we define v_0 and v_n such that $v_0 = v^{(0)}$ and $v_n = v^{(n)}$ to represent each program variable before and after the execution of the loop. Any

```
1       func main() {
2         Int[] a;
3         const Int[] b;
4         const Int length
5         Int i = 0;
6
7         while (i < length) {
8           a[i] = b[i]
9           i = i + 1;
10        }
11      }
12      assert(∀pos_𝕀.((0 ≤ pos < length) → a(end, pos) = b(pos)))
13
```

Figure 5.1: Program copying elements from array `b` to array `a`.

consequence that represents a loop invariant (1) should contain at least one target symbol or a skolem function (introduced during saturation), and (2) should only contain symbols that are either target symbols, skolem function or theory symbols, that is interpreted functions such as arithmetic symbols. Such symbols are called *useful*. Note that clauses containing only theory symbols are also eliminated as they are rather useless for inductive loop reasoning. While being valid consequences, they do not represent loop invariants and are, hence, eliminated.
To achieve finding such clauses during saturation, the internal ordering has to be adapted: by making all other *useless* symbols large in precedence, the prover applies inferences on "heavy" clauses first and thus eventually removes them from the search space. All consequences containing useful symbols can then be outputted as loop invariants. If such a clause contains a skolem function, it can be de-skolemized by reintroducing an existential quantifier. Appropriating saturation in such a way, therefore, enables finding quantified invariants potentially with quantifier alternations. We revisit this method in the context of trace logic $\mathcal{L}$.

5.2 Invariant Generation in Trace Logic

Reasoning in trace logic can also be exploited as an invariant generation engine, synthesizing first-order invariants using the Vampire theorem prover. In contrast to prior work, our program semantics in trace logic is more complex: rather than considering loop iterations as a single program state, we might have multiple program states per loop. Trace logic is an extension of extended expression where locations are not merely iterations, but timepoints - a cross product of locations and iterations. Consequently, we have to adapt the symbol-eliminating mode of Vampire to derive logical consequences of the trace logic semantics. Some of these consequences may be loop invariants. Intuitively, we generate consequences from our program semantics and trace lemmas, such that a conjunction of some of those consequences is strong enough to derive some safety asser-

tion. Given the quantified nature of our trace lemmas, this allows us to derive complex properties that may contain quantifier alternation. Additionally, we want to eliminate any intermediate timepoints from our consequences as we want to find formulas that are valid at any point in time. However, there is no guarantee that relevant consequences will be derived quickly or at all, as there may be an infinite number of consequences. Therefore, some heuristics must be applied to guide consequence finding for invariant synthesis.

The basic idea is aligned with [KV09]: loop invariants should only contain symbols from the input loop language, with no timepoints. We, thus, define symbols to be either *colored* or *transparent*, and eliminate *colored* symbols to obtain (quantified) formulas as conclusions containing only *transparent* symbols. Transparent symbols are either predefined target symbols, constant program variables that do not include any timepoints to begin with, or theory symbols. Note, however, that clauses containing only theory symbols are also eliminated as they are rather useless for inductive loop reasoning. While being valid consequences, they do not represent loop invariants and are, hence, eliminated. To the avid reader, it is evident that colored symbols are those representing program variables that are used in assignments. Specifically on the left-hand side of assignments since these are the program variables that change in value throughout computation and will, therefore, contain timepoints in their logical representation. To remove such constructs, we apply symbol elimination: any symbol representing a variable `v` used on the left-hand side of an assignment is eliminated. However, we still want to generate invariants containing otherwise-eliminated variables at specific locations, so for each eliminated variable `v` we define $\texttt{v\_init} = v(l_1)$ and $\texttt{v\_final} = v(l_2)$ for appropriate timepoints l_1, l_2: these new symbols need not be eliminated. Now, the most interesting timepoints for loop invariants are, of course, the first iteration $l_{\texttt{w}}(0)$ of some loop `w` since it represents the values of `v` before the execution of a loop, as well as the end of the loop execution represented by $l_{\texttt{w}}(nl_{\texttt{w}})$. That is for each loop `w` and each program variable `v` used the loop we define

$$\texttt{v\_init} = v(l_{\texttt{w}}(0))$$

and

$$\texttt{v\_final} = v(l_{\texttt{w}}(nl_{\texttt{w}}))$$

as *target symbols* and *color* the original program variable `v`.

Example 11 (Target symbols)**.** Consider the program in Figure 5.1 that copies elements from immutable array `b` to the mutable array `a` in a loop. Since only `a` and loop counter variable `i` appear on the left-hand side of assignments, we *color* these symbols and define their target symbols in the following way

1. $\forall x.\, \texttt{a\_init}(x) = a(l_7(0), x)$
2. $\forall x.\, \texttt{a\_final}(x) = a(l_7(nl_7), x)$
3. $\texttt{i\_init} = i(l_7(0))$
4. $\texttt{i\_final} = i(l_7(nl_7))$

Note that immutable symbols `b` and `length` as well as above defined target symbols `a_init`, `a_final`, `i_init` and `i_final` are *transparent* and hence can/should appear in generated consequences.

5.2.1 Refreshing Symbol Elimination

Theorem proving technology (and Vampire in particular) has changed and improved somewhat since the original work on this method. To restore the ability to extract useful invariants, the symbol elimination technique must be adapted to the current state of Vampire. Happily, most changes simply fix new Vampire features that did not consider symbol elimination.
Additionally, reasoning in trace logic and expressing trace lemmas for programs produces more complex encodings than in the original work, typically relying heavily on a theorem prover's ability to *simplify* consequences. The original approach yields candidate invariants as soon as undesirable symbols are eliminated, but this means candidates are not subject to the full set of simplifications Vampire offers. We further adjusted symbol elimination to output fully-simplified consequences during proof search in Vampire (the so-called *active set* [KV13]) at the end of a user-specified time limit. Consequences that contain colored symbols or are pure consequences of theories are removed at this stage. Our approach reduces the number of candidates produced per unit time, in exchange for "nicer" invariants that reflect all information available to the theorem prover.

5.2.2 Reasoning with Integers vs. Naturals

In the standard setting of reasoning with trace logic, we use natural numbers, that is terms of sort $\mathbb{N}$ to describe loop iterations. This benefits proof search as we limit the amount of theory axioms necessary to describe loop iterations to 0, `suc` and the $<$-relation. Omitting any arithmetic relieves the prover of unnecessary theory-based derivations that will not result in a proof.
However, in some situations it is advantageous to use the theory of integers $\mathbb{I}$: incremental loop counter variables `i` of sort $\mathbb{I}$ will have the same numerical value as nl of sort $\mathbb{N}$ at the end of a loop. The automated theorem prover cannot infer such reasoning when two different sorts are in use. We need integer-based loop iterations to allow deriving $i(l(nl)) = nl$, and finally `i_final` $= nl$ as a loop invariant by symbol-eliminating inferences.
Additionally, a clause such as $i(l(nl)) = nl$ can be very helpful for further consequence-finding. Let us consider the following property

$$\forall x_{\mathbb{I}}.\ 0 \leq x \leq length \rightarrow a(x) = b(x). \tag{5.1}$$

Assertion (5.1) essentially requires us to prove that two arrays `a, b` are equal in all positions between 0 and `length`. Such a property might for example be useful to prove when we copy from an array `b` into an array `a` in a loop with loop condition $i < length$ where `i` is the loop counter variable incremented by one in each iteration such as in Figure 5.1. Now, when we make use of the symbol elimination method, we might be

able to derive a property $\forall x.\ 0 \leq x < nl \rightarrow a(x) = b(x)$, essentially stating that the property holds for all iterations of the loop. Additionally, the prover can easily deduce that $\texttt{i\_final} \geq \mathit{length}$ for some loop counter variable `i` thanks to our semantics and thus conclude the validity of property (5.1).
However, in case of natural numbers $\mathbb{N}$ we cannot deduce that $i(l(nl)) = nl$ holds since the sorts of i and nl differ. Such a consequence is nonetheless required for the conjunction of generated invariants to be strong enough to prove postcondition (5.1). Consequently, we would depend upon the prover to discover the invariant $\forall x.\ 0 \leq x \leq \texttt{i\_final} \rightarrow a(x) = b(x)$ directly which cannot be deduced by the prover as our loop semantics are bounded by loop iterations rather than the loop counter values.
With integer-based loop iterations we can circumvent this problem as the prover finds the equality $i(l(nl)) = nl$ as a consequence which makes the conjunction of clauses strong enough to prove the desired postcondition.

Example 12 (Invariant generation with integers)**.** Consider again the program in Figure 5.1. Given a program semantics using integers as loop iterations and colored symbols `a` and `i`, we derive the following three invariants containing target symbols `i_final`, `a_final` and transparent symbols nl_7, *length* and b during *saturation with symbol elimination*:

(Inv1) $\texttt{i\_final} = nl_7$
(Inv2) $\texttt{i\_final} \geq \mathit{length}$
(Inv3) $\forall x.\ 0 \leq x < nl_7 \rightarrow \texttt{a\_final}(x) = b(x)$

It is easy to see that the conjunction of (Inv1)–(Inv3) is strong enough to prove the safety assertion containing target symbols:

$$\forall x. 0 \leq x < \mathit{length} \rightarrow \texttt{a\_final}(x) = b(x).$$

By definition of the target symbols, it follows that property (5.1) holds for the program in Figure 5.1.

5.3 Related Work

Loop invariant synthesis is a highly active research field. A wide range of approaches to loop analysis and (quantified) invariant generation has been developed in recent years. However, loop invariant synthesis over array-transforming loops becomes very challenging as complex quantification might be involved. Most state-of-the-art approaches that produce quantified invariants are based on SMT-solving such as [GSM16, KBI+17], thus by nature of the underlying solvers restricted to universal quantification. The work [KBI+17] is based on iteratively generating quantifier-free properties and lift them to universally quantified invariants.
When it comes to generating invariants for programs containing unbounded data structures, we have to most notably mention the constrained Horn clause (CHC) solvers `Spacer/Quic3` [KCSG20, GSV18] and `FreqHorn` [FPMG19]. These approaches are based on IC3/PDR [HB12], for the former, and sampling/enumerating invariants until a

conjunction of generated formulas is inductive, for the latter. Given that these approaches heavily rely on SMT-solvers to verify the validity of invariants with regards to a program and a safety assertion, they are mostly limited to universally quantified invariants if quantification is supported at all.
Another line of research is based on invariant templates with universal quantification. In [BMR13], quantifier-free invariants are computed and lifted by universally quantified templates. A similar approach is used in [LRCR13]. Both of these approaches use SMT-solving and thus suffer from the same limitations.
Another approach based on enumerating formulas as invariant candidates can be found in [MPMW20, PSM16]. Specifically, [MPMW20] generates so-called representation invariants that synthesize invariants over the values of recursive datatypes by enumerating and alternatingly weakening and strengthening of invariant candidates. However the input language is restricted to universally quantified specifications.
Similarly, [YTGN22] can establish quantified invariants for distributed protocols based on sampling distributed protocol states at different instance sizes and enumerating the strongest possible invariants for these samples. To check inductiveness of the candidate invariants an SMT-solver is employed, hence candidates are iteratively weakened making invariant discovery efficient. However, contrary to our approach the SMT-solving bottleneck on existential quantification applies.
While most works suffer from the limitations of SMT-solving, there are some approaches that can handle existential or alternating quantification. The work of [KPIA20] extends the IC3/PDR algorithm with so-called quantified separators, allowing them to generate non-trivial formulas automatically. While it is unclear whether their approach works on programs with loops and unbounded data structures, they could establish some invariants with quantifier alternations for distributed protocols in their approach. Further, [ZCF23] provides a novel approach of CHC-solving for recursive datatypes with by generating recursive functions. They employ first-order theorem provers rather than relying on SMT to check the validity of their candidate functions. However, given the universally quantified nature of algebraic datatypes' inductive definitions, it is indefinite whether queries with existential or alternating quantification can be solved.

CHAPTER 6

Computation Induction for Recursive Sorting Algorithms

This chapter is based on our work

> Pamina Georgiou, Marton Hajdu, and Laura Kovács. Sorting Without Sorts. No. 10632. EasyChair Preprint, 2023. *Currently under submission.*

Sorting algorithms are integrated parts of any modern programming language, hence ubiquitous in computing, which naturally triggers the demand of validating the functional correctness of sorting routines. They typically implement recursive/iterative operations over potentially unbounded data structures, for instance lists or arrays, combined with arithmetic manipulations of numeric data types, such as naturals, integers or reals. Automating the formal verification of sorting routines, therefore, brings the challenge of automating recursive/inductive reasoning in extensions and combinations of first-order theories, while also addressing the reasoning burden arising from design choices made for the purpose of efficient sorting. Most notably, `Quicksort` [Hoa62] is known to be easily implemented when making use of recursive function calls, for example, as given in Figure 6.1, whereas procedural implementations of `Quicksort` would require additional recursive data structures such as stacks. While `Quicksort` and other sorting routines have been proven correct by means of manual efforts [FH71], proof assistants [NBE+21, WS04, BSSU17], abstract interpreters [GMT08], or model checkers [JM07], to the best of our knowledge such correctness proofs so far have not been fully automated.

In this chapter we aim to verify the partial correctness of functional programs with recursive data structures, in an automated manner by using saturation-based first-order theorem proving. To achieve this, we turn the automated first-order reasoner into a complementary approach to interactive proof assistance: (i) we rely on manual guidance in splitting inductive proof goals into subgoals (Sections 6.4 and 6.5), but (ii) fully automate inductive proofs in saturation-based reasoning (Section 6.3).

```
1  datatype  a' list = nil | cons(a', (a' list))
2
3  quicksort :: a' list → a' list
4  quicksort(nil) = nil
5  quicksort(cons(x, xs)) =
6    append(
7      quicksort(filter<(x, xs)) ,
8      cons(x, quicksort(filter≥(x, xs))))
9
10 append :: a' list → a' list → a' list
11 append(nil, xs) = xs
12 append(cons(x, xs), ys) = cons(x, append(xs, ys))
13
```

Figure 6.1: Recursive functional algorithm of `Quicksort`, using the recursive function definitions `append`, $\texttt{filter}_<$ and $\texttt{filter}_\geq$ over lists of sort a. The datatype *list* is inductively defined by the list constructors nil and `cons`. Here, xs, ys denote lists whose elements are of sort a, whereas x is a list element of sort a. The `append` function concatenates two lists. The $\texttt{filter}_<$ and $\texttt{filter}_\geq$ functions return lists of elements y of xs such that $y < x$ and $y \geq x$, respectively.

The crux of our approach is a compositional reasoning setting based on superposition-based first-order theorem proving [KV13] with native support for induction [HHK+22] and first-order theories of recursively defined data types [KRV17]. We extend this setting to support the first-order theory of list data structures parameterized by an abstract background theory/sort a and advocate *computation induction* for induction on recursive function calls. As such, our framework allows us to automatically discharge manually split verification conditions that require inductive proofs, without requiring manually proven or a priori given inductive annotations such as loop invariants, nor user input to perform proofs by induction. Doing so, we automatically derive induction axioms during *saturation* to establish the functional correctness of the recursive implementation of `Quicksort` from Figure 6.1 by means of automated first-order reasoning. In a nutshell, we proceed as follows.

(i) We formalize the *semantics of functional programs* in extensions of the first-order theory of lists (Section 6.2). Rather than focusing on lists with a specific background theory, such as integers/naturals, our formalization relies on a parameterized sort/type a abstracting specific (arithmetic) theories. To this end, we impose that the sort a has a linear order $\leq$. We then express program semantics in the first-order theory of lists parameterized by a, allowing us to quantify over lists of sort a as they are domain elements of our first-order theory.
Doing so, we remark that one of the major reasoning burdens towards establishing the correctness of sorting algorithms comes with formalizing permutation properties, for example that two lists are permutations of each other. Universally quantifying over permutations of lists is, however, not a first-order property and hence reasoning about list permutation requires higher-order logic. While counting and comparing the number

of list elements is a viable option to formalize permutation equivalence in first-order logic, the necessary arithmetic reasoning adds an additional burden to the underlying prover. We overcome this challenge by introducing an effective first-order formalization of *permutation equivalence* over parameterized lists. Our permutation equivalence property encodes *multiset* operations over lists, eliminating the need of counting list elements, and therefore arithmetic reasoning, or fully axiomatizing (higher-order) permutations.

(ii) We revise inductive reasoning in first-order theorem proving (Section 6.3) and introduce *computation induction* as a means to tackle recursive divide-and-conquer algorithms. We, therefore, extend the first-order reasoner with an inductive inference based on the *computation induction scheme* and outline its necessity for recursive sorting routines.

(iii) We leverage *first-order theorem proving for compositional proofs* of recursive parameterized sorting algorithms (Section 6.4), in particular of `Quicksort` from Figure 6.1. By embedding the application of induction directly in saturation-based proving, we automatically discharge manually split proof obligations. Each such condition represents a first-order lemma, and hence a proof step. We emphasize that the only manual effort in our framework comes with splitting formulas into multiple lemmas (Section 6.5.1); each lemma is established automatically by means of automated theorem proving with built-in induction. That is, all our lemmas/verification conditions are automatically proven by means of structural and/or computation induction during the saturation process. Thanks to the automation of induction in saturation, we turn first-order theorem proving into a powerful approach to guide human reasoning about recursive properties. We do not rely on user-provided inductive properties, nor on user guidance to perform proofs by induction, but generate inductive hypotheses/invariants via inductive inferences automatically as logical consequences of our program semantics.

(iv) We note that sorting algorithms often follow a divide-and-conquer approach (see Figure 6.2). We, thus, apply our approach on other sorting routines and investigate a generalized set of manual proof splits/lemmas that is applicable to verify functional sorting algorithms on recursive data structures (Section 6.5) and guides compositional reasoning in saturation-based theorem proving for this purpose.

Our work is implemented in the Vampire theorem prover [KV13]. Details are provided in Section 7.4.2. We demonstrate our findings with an experimental evaluation in Chapter 8, Section 8.2.

6.1 Background

Parameterized Lists. We use the first-order theory of recursively defined datatypes [KRV17]. In particular, we consider the list datatype with two constructors nil and $\mathsf{cons}(x, xs)$, where nil is the empty list and x and xs are respectively the head and tail of a list. We introduce a type parameter a that abstracts the sort/background theory of the list elements. Here, we impose the restriction that the sort a has a linear order $<$, that is, a binary relation which is reflexive, antisymmetric, transitive and total. For simplicity,

we also use $\geq$ and $\leq$ as the standard ordering extensions of $<$. As a result, we work in the first-order theory of lists parameterized by sort a, allowing us to quantify over lists as domain elements of this theory. For simplicity, we write xs_a, ys_a, zs_a to mean that the lists xs, ys, zs are parameterized by sort a; that is their elements are of sort a. Similarly, we use x_a, y_a, z_a to mean that the list elements x, y, z are of sort a. Whenever it is clear from the context, we omit specifying the sort a.

Function definitions. We make the following abuse of notation. For some function `f` in some program `P`, we use the notation `f(arg`$_1$`, ...)` to refer to function definitions/calls appearing in the input algorithm, while the mathematical notation $f(arg_1, ...)$ refers to its counterpart in our logical representation, that is the function call semantics in first-order notation as introduced in Section 6.2.

6.2 First-Order Semantics of Functional Sorting Algorithms

We outline our formalization of recursive sorting algorithms in the full first-order theory of parameterized lists.

6.2.1 Recursive Functions in First-Order Logic

We investigate recursive algorithms written in a functional coding style and defined over lists using list constructors. That is, we consider recursive functions `f` that manipulate the empty list nil and/or the list cons(x, xs).
Many recursive sorting algorithms, as well as other recursive operations over lists, implement a *divide-and-conquer* approach: let `f` be a function following such a pattern, `f` uses (i) a *partition function* to divide $list_a$, that is a *list* of sort a, into two smaller sublists upon which `f` is recursively applied to, and (ii) calls a *combination function* that puts together the result of the recursive calls of `f`. Figure 6.2 shows such a divide-and-conquer pattern, where the partition function `partition` uses an invertible operator $\circ$, with $\circ^{-1}$ being the inverse of $\circ$; `f` is applied to the results of $\circ$ before these results are merged using the combination function `combine`.

Note that the recursive function `f` of Figure 6.2 is defined via the declaration $f :: a'list \to ... \to a'list$, where ... denotes further input parameters. We formalize the first-order semantics of `f` via the function $f\colon (list_a \times ...) \mapsto list_a$, by translating the inductive function definitions `f` to the following first-order formulas with parameterized lists (in first-order logic, function definitions can be considered as universally quantified equalities):

```
1  f ::  a' list → ... → a' list
2  f(nil, ...)  =  nil
3  f(cons(y, ys), ...) =
4    combine(
5      f(partition∘(cons(y,ys))),
6      f(partition∘⁻¹(cons(y,ys)))
7    )
8
```

Figure 6.2: Recursive divide-and-conquer approach.

$$\begin{array}{rcl} f(\mathsf{nil}) & = & \mathsf{nil} \\ \forall x_a, xs_a \,.\, f(\mathsf{cons}(x, xs)) & = & combine(\quad f(partition_{\circ}(\mathsf{cons}(x, xs))), \\ & & \qquad\quad f(partition_{\circ^{-1}}(\mathsf{cons}(x, xs)))). \end{array} \tag{6.1}$$

The recursive divide-and-conquer pattern of Figure 6.2, together with the first-order semantics (6.1) of `f`, will be respectively used in Sections 6.4-6.5 for proving correctness of the `Quicksort` algorithm (and other sorting algorithms), as well as for applying lemma generalizations for divide-and-conquer list operations. We next introduce our first-order formalization for specifying that `f` implements a sorting routine.

6.2.2 First-Order Specification of Sorting Algorithms

We consider a specific function instance of `f` implementing a sorting algorithm, expressed through $sort :: a'list \rightarrow a'list$. The functional behavior of *sort* needs to satisfy two specifications implying the functional correctness of *sort*: (i) sortedness and (ii) permutation equivalence of the list computed by *sort*.

(i) Sortedness: *The list computed by the sort function must be sorted w.r.t. some linear order* $\leq$ *over the type a of list elements.* We define a parameterized version of this sortedness property using an inductive predicate *sorted* as follows:

$$\begin{array}{rcl} sorted(\mathsf{nil}) & = & \top \\ \forall x_a, xs_a \,.\, sorted(\mathsf{cons}(x, xs)) & = & (elem_{\leq}list(x, xs) \wedge sorted(xs)), \end{array} \tag{6.2}$$

where $elem_{\leq}list(x, xs)$ specifies that $x \leq y$ for any element y in xs. Proving correctness of a sorting algorithm *sort* thus reduces to proving the validity of:

$$\forall xs_a \,.\, sorted(sort(xs)). \tag{6.3}$$

(ii) Permutation Equivalence: *The list computed by the sort function is a permutation of the input list to the sort function.* In other words the input and output lists of *sort* are permutations of each other, in short permutation equivalent.

```
1 filter_Q :: a' → a' list → a' list
2 filter_Q (x, nil)  = nil
3 filter_Q (x, cons (y, ys) ) =
4   if (Q(y,x)) {
5     cons (y,filter_Q (x,ys) )
6   } else {
7     filter_Q(x,ys)
8   }
9
```

Figure 6.3: Function $filter_Q$ filtering elements of a list, by using a predicate $Q(y, x)$ over list elements x, y.

Axiomatizing permutations requires quantification over relations and is thus not expressible in first-order logic [LM96]. A common approach to prove permutation equivalence of two lists is to count the occurrence of elements in each list respectively and compare the occurrences of each element. Yet, counting adds a burden of arithmetic reasoning over naturals to the underlying prover, calling for additional applications of mathematical induction. We overcome these challenges of expressing permutation equivalence as follows. We introduce a family of functions $filter_Q$ manipulating lists, with the function $filter_Q$ being parameterized by a predicate Q and as given in Figure 6.3.
In particular, given an element x and a list ys, the functions $filter_=$, $filter_<$, and $filter_\geq$ compute the maximal sublists of ys that contain only equal, resp. smaller and greater-or-equal elements to x. Analogously to counting the multiset multiplicity of x in ys via counting functions, we compare lists given by $filter_=$, avoiding the need to count the number of occurrences of x and hence prolific axiomatizations of arithmetic. Thus, to prove that the input/output lists of *sort* are permutation equivalent, we show that, for every list element x, the results of applying `filter`$_=$ to the input/output list of *sort* are the same over all elements. Formally, we have the following first-order property of permutation equivalence:

$$\forall x_a, xs_a \,.\, filter_=(x, xs) = filter_=(x, sort(xs)). \tag{6.4}$$

6.3 Computation Induction in Saturation

In this section, we describe our reasoning extension to saturation-based first-order theorem proving, in order to support inductive reasoning for recursive sorting algorithms as introduced in Section 6.2. Our key reasoning ingredient comes with a structural induction schemata of *computation induction*, which we directly integrate in the saturation proving process.
We revisited the structural induction schema over lists in Section 2.2. Sorting algorithms, however, often require induction axioms that are more complex than instances of structural induction (2.1). Such axioms are typically instances of computation/recursion

induction schema, arising from divide-and-conquer strategies as introduced in Section 6.2.1. Particularly, the complexity arises due to the two recursive calls on different parts of the original input list produced by the *partition* function that have to be taken into account by the induction schema. We therefore use the following *computation induction* schema over lists:

$$\Big(L[\mathsf{nil}] \wedge \forall x, ys. \Big(\begin{pmatrix} L[partition_{\circ}(x, ys)] \wedge \\ L[partition_{\circ^{-1}}(x, ys)] \end{pmatrix} \rightarrow L[\mathsf{cons}(x, ys)]\Big)\Big) \rightarrow \forall zs. L[zs] \tag{6.5}$$

yielding the following instance of the Ind inference rule that can be applied by the prover during saturation:

$$\frac{\overline{L}[t] \vee C}{\begin{array}{c} \overline{L}[\mathsf{nil}] \vee L[partition_{\circ}(\sigma_x, \sigma_{ys})] \vee C \\ \overline{L}[\mathsf{nil}] \vee L[partition_{\circ^{-1}}(\sigma_x, \sigma_{ys})] \vee C \\ \overline{L}[\mathsf{nil}] \vee \overline{L}[\mathsf{cons}(\sigma_x, \sigma_{ys})] \vee C \end{array}}$$

where t is a ground term of sort list, $L[t]$ is ground, σ_x and σ_{ys} are fresh constant symbols, and $partition_{\circ}$ and its inverse refer to the functions that partition lists into sublists within the actual actual sorting algorithms. Note that the above Ind inference instance results in three clauses.
In the following, we show how instances of the Ind inference rule with schemes (2.1) and (6.5) are leveraged to automatically prove sortedness and permutation equivalence over sorting routines by splitting proof obligations into multiple first-order lemmas.

6.4 Proving Recursive Quicksort

We now describe our approach towards proving the correctness of the recursive parameterized version of Quicksort, as given in Figure 6.1. Note that Quicksort recursively sorts two sublists that contain respectively smaller and greater-or-equal elements than the pivot element x of its input list. We therefore reduce the task of proving the functional correctness of Quicksort to the task of proving the (i) sortedness property (6.3) and (ii) the permutation equivalence property (6.4) of Quicksort. As mentioned in Section 6.2.2, a similar reasoning is needed for most sorting algorithms, which we evidence in Section 6.5, as well as in our experimental evaluation (Section 8.2).

6.4.1 Proving Sortedness for Quicksort

Given an input list xs, we prove that Quicksort computes a sorted list by considering the property (6.3) instantiated for Quicksort. That is, we prove:

$$\forall xs_a \,.\, sorted(quicksort(xs)) \tag{6.6}$$

The sortedness property (6.6) of Quicksort is proved via *compositional reasoning* over (6.6). Namely, we enforce the following two properties that together imply (6.6):

(S1) By using the linear order $\leq$ of the background theory a, for any element y in the sorted list $quicksort(filter_{<}(x, xs))$ and any element z in the sorted list $quicksort(filter_{\geq}(x, xs))$, we have $y \leq x \leq z$.

(S2) The functions $filter_{<}$ and $filter_{\geq}$ of Figure 6.3 are correct. That is, filtering elements from a list that are smaller, respectively greater-or-equal, than an element x results in sublists only containing elements smaller than, respectively greater-or-equal, than x.

Similarly to (6.2), in order to express property **(S2)** we introduce the predicates $elem_{\leq}list :: a' \to a'list \to bool$ and $list_{\leq}list :: a'list \to a'list \to bool$, defined inductively as:

$$\begin{aligned} \forall x_a \,.\, elem_{\leq}list(x, \mathsf{nil}) &= \top \\ \forall x_a, y_a, ys_a \,.\, elem_{\leq}list(x, \mathsf{cons}(y, ys)) &= x \leq y \wedge elem_{\leq}list(x, ys), \end{aligned} \tag{6.7}$$

and

$$\begin{aligned} \forall ys_a \,.\, list_{\leq}list(\mathsf{nil}, ys) &= \top \\ \forall x_a, xs_a, ys_a \,.\, list_{\leq}list(\mathsf{cons}(x, xs), ys) &= (elem_{\leq}list(x, ys) \wedge list_{\leq}list(xs, ys)). \end{aligned} \tag{6.8}$$

That is, for some element x and lists xs, ys, we express that x is smaller than or equal to any element of xs by $elem_{\leq}list(x, xs)$. Similarly, $list_{\leq}list(xs, ys)$ captures that every element in list xs is smaller than or equal to any element in ys.
The inductively defined predicates of (6.7)–(6.8) allow us to express necessary lemmas over list operations preserving the sortedness property (6.6), for example, to prove that appending sorted lists yields a sorted list.
Proving properties **(S1)**–**(S2)**, and hence deriving the sortedness property (6.6) of `Quicksort`, requires *three first-order lemmas* in addition to the first-order semantics (6.1) of `Quicksort`. Each of these lemmas is automatically proven by saturation-based theorem proving using the structural and/or computation induction schemata of (2.1) and (6.5); hence, by compositionality, we obtain **(S1)**–**(S2)** implying (6.6). We next discuss these three lemmas and outline the complete (compositional) proof of the sortedness property (6.6) of `Quicksort`.
• In support of **(S1)**, lemma (6.9) expresses that for two *sorted* lists xs, ys and a list element x, such that $elem_{\leq}list(x, xs)$ holds and all elements of the constructed list $\mathsf{cons}(x, xs)$ are greater than or equal to all elements in ys, the result of concatenating ys and $\mathsf{cons}(x, xs)$ yields a sorted list. Formally, we have

$$\begin{aligned} \forall x_a, xs_a, ys_a \,. \quad & (sorted(xs) \wedge sorted(ys) \wedge elem_{\leq}list(x, xs) \wedge \\ & list_{\leq}list(ys, \mathsf{cons}(x, xs))) \\ & \to sorted(append(ys, \mathsf{cons}(x, xs))) \end{aligned} \tag{6.9}$$

• In support of **(S2)**, we need to establish that filtering greater-or-equal elements for some list element x results in a list whose elements are greater-or-equal than x. In other words, the inductive predicate of (6.7) is invariant over sorting and filtering operations over lists.

$$\forall x_a, xs_a \,.\, elem_{\leq}list(x, quicksort(filter_{\geq}(x, xs))). \tag{6.10}$$

• Lastly and in further support of **(S1)**–**(S2)**, we establish that all elements of a list xs are "covered" with the filtering operations $\texttt{filter}_{\geq}$ and $\texttt{filter}_{<}$ w.r.t. a list element x of xs. Intuitively, a call of $\texttt{filter}_{<}\texttt{(x,xs)}$ results in a list containing all elements of xs that are smaller than x, while the remaining elements of xs are those that are greater-or-equal than x and hence are contained in $\mathsf{cons}(x, filter_{\geq}(x, xs))$. By applying Quicksort over the input list xs, we thus have:

$$\begin{array}{l}\forall x_a, xs_a\,.\\ \quad list_{\leq}list(quicksort(filter_{<}(x, xs)), \mathsf{cons}(x, quicksort(filter_{\geq}(x, xs)))).\end{array} \tag{6.11}$$

The first-order lemmas (6.9)–(6.11) guide saturation-based proving to instantiate structural/computation induction schemata and derive the following induction axiom necessary to prove **(S1)**–**(S2)**, and hence sortedness of Quicksort:

$$\begin{array}{l}\Big(sorted(quicksort(\mathsf{nil}))\wedge\\ \quad \forall x_a, xs_a\,.\,\left(\begin{array}{l} sorted(quicksort(filter_{\geq}(x, xs)))\wedge\\ sorted(quicksort(filter_{<}(x, xs)))\end{array}\right) \rightarrow sorted(quicksort(\mathsf{cons}(x, xs))\Big)\\ \rightarrow \forall xs_a\,.\,sorted(quicksort(xs)),\end{array} \tag{6.12}$$

where axiom (6.12) is automatically obtained during saturation from the computation induction schema (6.5). Intuitively, the prover replaces F by $sorted(quicksort(t))$ for some term t, and uses $filter_{<}$ and $filter_{\geq}$ as $partition_{\circ}$ and $partition_{\circ^{-1}}$ respectively to find the necessary computation induction scheme. We emphasize that this step is fully automated during the saturation run.

The first-order lemmas (6.9)–(6.11), together with the induction axiom (6.12) and the first-order semantics (6.1) of Quicksort, imply the sortedness property (6.4) of Quicksort; this proof can automatically be derived using saturation-based reasoning. Yet, the obtained proof assumes the validity of each of the lemmas (6.9)–(6.11). To eliminate this assumption, we propose to also prove lemmas (6.9)–(6.11) via saturation-based reasoning. Yet, while lemma (6.9) is established by saturation with structural induction (2.1) over lists, proving lemmas (6.10)–(6.11) requires further first-order formulas. In particular, for proving lemmas (6.10)–(6.11) via saturation, we use four further lemmas, as follows.

• Lemmas (6.13)–(6.14) indicate that the order of $elem_{\leq}list$ and $list_{\leq}list$ is preserved under $quicksort$, respectively. That is,

$$\forall x_a, xs_a\,.\,elem_{\leq}list(x, xs) \rightarrow elem_{\leq}list(x, quicksort(xs)) \tag{6.13}$$

and

$$\forall xs_a, ys_a\,.\,list_{\leq}list(ys, xs) \rightarrow list_{\leq}list(quicksort(ys), xs). \tag{6.14}$$

• Proving lemmas (6.13)–(6.14), however, requires two further lemmas that follow from saturation with built-in computation and structural induction, respectively. Namely, lemmas (6.15)–(6.16) establish that $elem_{\leq}list$ and $list_{\leq}list$ are also invariant over appending lists. That is,

$$\begin{array}{ll}\forall x_a, y_a, xs_a, ys_a\,. & (y \leq x \wedge elem_{\leq}list(y, xs) \wedge elem_{\leq}list(y, ys))\\ & \rightarrow elem_{\leq}list(y, append(\mathsf{cons}(x, ys), xs))\end{array} \tag{6.15}$$

and

$$\forall xs_a, ys_a, zs_a.\quad \begin{array}{l}(list_{\leq}list(ys, xs) \wedge list_{\leq}list(zs, xs))\\ \rightarrow list_{\leq}list(append(ys, zs), xs)\end{array} \tag{6.16}$$

With lemmas (6.13)–(6.16), we automatically prove lemmas (6.9)–(6.11) via saturation-based reasoning. The complete automation of proving properties **(S1)**–**(S2)**, and hence deriving the sortedness property (6.6) of `Quicksort` in a compositional manner, requires thus *altogether seven lemmas* in addition to the first-order semantics (6.1) of `Quicksort`. Each of these lemmas is automatically established via saturation with built-in induction. Hence, unlike interactive theorem proving, compositional proving with first-order theorem provers can be leveraged to eliminate the need to a priori specifying necessary induction axioms to be used during proof search.

6.4.2 Proving Permutation Equivalence for `Quicksort`

In addition to establishing the sortedness property (6.6) of `Quicksort`, the functional correctness of `Quicksort` also requires proving the permutation equivalence property (6.4) for `Quicksort`. That is, we prove:

$$\forall x_a, xs_a.\ filter_{=}(x, xs) = filter_{=}(x, quicksort(xs)). \tag{6.17}$$

In this respect, we follow the approach introduced in Section 6.2.2 to enable first-order reasoning over permutation equivalence (6.17). Namely, we use $filter_{=}$ to filter elements x in the lists xs and $quicksort(xs)$, respectively, and build the corresponding multisets containing as many x as x occurs in xs and $quicksort(xs)$. By comparing the resulting multisets, we implicitly reason about the number of occurrences of x in xs and $quicksort(xs)$, yet, without the need to explicitly count occurrences of x. In summary, we reduce the task of proving (6.17) to *compositional reasoning* again, namely to proving following *two properties given as first-order lemmas* which, by compositionality, imply (6.17):

(P1) List concatenation commutes with $filter_{=}$, expressed by the lemma:

$$\forall x_a, xs_a, ys_a.\ filter_{=}(x, append(xs, ys)) = append(\begin{array}{l}filter_{=}(x, xs),\\ filter_{=}(x, ys)).\end{array} \tag{6.18}$$

(P2) Appending the aggregate of both `filter`-operations results in the same multisets as the unfiltered list, that is, permutation equivalence is invariant over combining inverse reduction operations. This property is expressed via lemma:

$$\forall x_a, y_a, xs_a.\ filter_{=}(x, xs) = append(\begin{array}{l}filter_{=}(x, filter_{<}(y, xs)),\\ filter_{=}(x, filter_{\geq}(y, xs))).\end{array} \tag{6.19}$$

Similarly as in Section 6.4.1, we prove lemmas (6.18)–(6.19) by saturation-based reasoning with built-in induction. In particular, investigating the proof output shows that lemma (6.18) is established using the structural induction schema (2.1) in saturation, while the validity of lemma (6.19) is obtained by applying the computation induction schema (6.5) in saturation.

By proving lemmas (6.18)–(6.19), we thus establish validity of permutation equivalence (6.17) for `Quicksort`. Together with the sortedness property (6.6) of `Quicksort` proven in Section 6.4.1, we conclude the functional correctness of `Quicksort` in a compositional manner, using automated saturation-based theorem proving with built-in induction and *altogether nine first-order lemmas* in addition to the first-order semantics (6.1) of `Quicksort`.

6.5 Lemma Generalizations for Guided Proof Splits

Establishing the functional correctness of `Quicksort` in Section 6.4 uses nine first-order lemmas that express inductive properties over lists in addition to the first-order semantics (6.1) of `Quicksort`. While each of these lemmas is proved by saturation using structural/computation induction schemata, coming up with proper inductive lemmas remains crucial in reasoning about inductive data structures. That is, we have to find effective ways to split the proof such that the first-order theorem prover can automatically discharge all proof steps with built-in induction.
In Section 6.5.1, we describe when and how we split proof obligations into lemmas, so that each of these lemmas can further be proved automatically using first-order theorem proving. In Section 6.5.2, we next demonstrate that the lemmas of Section 6.4 can be generalized and leveraged to prove correctness of other divide-and-conquer list sorting algorithms, in particular within the `Mergesort` routine of Figure 6.5. The generality of our inductive lemmas from Section 6.4 also helps reasoning about sorting routines that do not necessarily follow a divide-and-conquer strategy, such as the `Insertionsort` algorithm of (Figure 6.4).

6.5.1 Guided Proof Splitting

Contrary to automated approaches that use inductive annotations to alleviate inductive reasoning, our approach synthesizes the correct induction axioms automatically during saturation runs to prove properties and lemmas correct. However, a manual limitation remains, namely proof splitting. That is, deciding when a lemma is necessary or helpful for the automated reasoner.
Splitting the proof into multiple lemmas is necessary to guide the prover to find the right terms to apply the inductive inferences of Section 6.3. This is particularly the case when input problems, such as sorting algorithms, contain calls to multiple recursive functions - each of which has to be shown to preserve the property that is to be verified.
In the following, we illustrate and examine the need for proof splitting using lemma (6.9).

Example 13 (Compositional reasoning over sortedness in saturation)**.** Consider the following stronger version of lemma (6.9) in the proof of `Quicksort`:

$$\forall x_a, xs_a, ys_a \,.\; (sorted(xs) \wedge sorted(ys)) \rightarrow sorted(append(ys, \mathsf{cons}(x, xs))). \tag{6.20}$$

This formula could automatically be derived by saturation with computation induction (6.5) while trying to prove sortedness of the algorithm. However, formula (6.20) is not valid with regards to the specification of `Quicksort` since the value of x is not correctly restricted w.r.t. $\leq$ to xs, ys (e.g. concatenating a sorted xs with an arbitrary x not necessarily yields a sorted list). Thus, the prover needs additional information to verify sortedness. Therefore, the assumptions $elem_{\leq} list(x, xs)$ and $list_{\leq} list(ys, \mathsf{cons}(x, xs))$ are needed in addition to (6.20), resulting in lemma (6.9). Yet, lemma (6.9) from Section 6.4 can be automatically derived via saturation with *compositional reasoning*, based on computation induction (6.5). That is, we manually split proof obligations based on missing information in the saturation runs: we derive (6.20) from (6.5) via saturation, strengthen the hypotheses of (6.20) with missing necessary conditions $elem_{\leq} list(x, xs)$ and $list_{\leq} list(ys, \mathsf{cons}(x, xs))$, and prove their validity via saturation, thus yielding (6.9).

Discussion. Contrary to loop invariants or other inductive annotations, our approach inductively proves each lemma correct by synthesizing the correct induction axioms during proof search fully automatically. In case a proof fails, we investigate the synthesized induction axioms, manually strengthen the property and add any additional assumptions as proof obligations whose validity is in turn again verified with the theorem prover and built-in induction. That is, we do not simply assume inductive lemmas but also provide a formal argument of their validity. We emphasize that we manually split the proof into multiple verification conditions such that inductive reasoning can be automated in saturation.

6.5.2 Lemma Generalizations for Sorting

The lemmas from Section 6.4 represent a number of common proof splits that can be applied to various list sorting tasks. In the following we generalize their structure and apply them to two other sorting algorithms, namely `Mergesort` and `Insertionsort`.

Common Patterns of Inductive Lemmas for Sorting Algorithms. Consider the computation induction schema (6.5). When using (6.5) for proving the sortedness (6.6) and permutation equivalence (6.17) of `Quicksort`, the inductive formula F of (6.5) is, respectively, instantiated with the predicates *sorted* from (6.6) and $filter_{=}$ from (6.17). The base case $F[\mathsf{nil}]$ of schema (6.5) is then trivially proved by saturation for both properties (6.6) and (6.17) of `Quicksort`.
Proving the induction step case of schema (6.5) is however challenging as it relies on *partition*-functions which are further used by *combine* functions within the divide-and-conquer patterns of Figure 6.2. Intuitively this means, that proving the induction step case of schema (6.5) for the sortedness (6.6) and permutation equivalence (6.17) properties requires showing that applying *combine* functions over *partition* functions preserve sortedness (6.6) and permutation equivalence (6.17), respectively. For divide-and-conquer algorithms of Figure 6.2, the step case of schema (6.5) requires thus proving

```
1  insertsort :: a' list → a' list
2  insertsort(nil) = nil
3  insertsort(cons(x, xs)) = insert(x, insertsort(xs))
4
5  insert :: a' → a' list → a' list
6  insert(x, nil) = cons(x, nil)
7  insert(x, cons(y, ys)) =
8    if (x ≤ y) {
9      cons(x, cons(y, ys))
10   } else {
11     cons(y, insert(x, ys))
12   }
13
```

Figure 6.4: Recursive algorithm of `Insertionsort` using the recursive function definition `insertsort` and auxiliary (recursive) function `insert`. `Insertionsort` recursively sorts the list by inserting single elements in the correct order with the helper function `insert`.

the following lemma:

$$\left(\forall x_a, ys_a.\left(combine\begin{pmatrix}L[partition_{\circ}(x,ys)],\\ L[partition_{\circ^{-1}}(x,ys)]\end{pmatrix} \rightarrow L[\mathsf{cons}(x,ys)]\right)\right). \tag{6.21}$$

To do so, we next describe generic instances of lemmas to be used in proving such step cases and hence functional correctness of sorting algorithms, depending on the *partition*/*combine* function of the underlining divide-and-conquer sorting routine.

(i) ***Combining sorted lists preserves sortedness.*** For proving the inductive step case (6.21) of the sortedness property (6.3) of sorting algorithms, we require the following generic lemma (6.3):

$$\forall xs_a, ys_a.\ (sorted(xs) \wedge sorted(ys)) \rightarrow sorted(combine(xs, ys)), \tag{6.22}$$

ensuring that combining sorted lists results in a sorted list. Lemma (6.22) is used to establish property **(S1)** of `Quicksort`, namely used as lemma (6.9) for proving the preservation of sortedness under the *append* function.
We showcase that generality of lemma (6.22), by using it upon sorting routines different than `Quicksort`. Consider, for example, `Mergesort` as given in Figure 6.5. The sortedness property (6.3) of `Mergesort` can be proved by using saturation with lemma (6.22); note that the `merge` function of `Mergesort` acts as a *combine* function of (6.22). That is, we establish the sortedness property of `Mergesort` via the following instance of (6.22):

$$\forall xs_a, ys_a.\ sorted(xs) \wedge sorted(ys) \rightarrow sorted(merge(xs, ys)) \tag{6.23}$$

```
1  mergesort :: a' list → a' list
2  mergesort(nil) = nil
3  mergesort(xs) =
4    merge(
5      mergesort(take((xs_length div 2), xs)) ,
6      mergesort(drop((xs_length div 2), xs))
7    )
8
9  merge :: a' list → a' list → a' list
10 merge(nil, ys) = ys
11 merge(xs, nil) = xs
12 merge(cons(x, xs), cons(y, ys)) =
13   if (x ≤ y) {
14     cons(x, merge(xs, cons(y, ys)))
15   } else {
16     cons(y, merge(cons(x, xs), ys))
17   }
18
```

Figure 6.5: Recursive `Mergesort` using the recursive functions `merge`, `take`, and `drop` over lists of sort a. `Mergesort` splits the input list xs into two halves by using `take` and `drop` that, respectively, *take* and *drop* the first half of elements of the input list (corresponding to `partition` functions of Figure 6.2). Both halves are recursively sorted and combined by the `merge` function, yielding a sorted list (corresponding to `combine` of Figure 6.2).

Finally, lemma (6.22) is not restricted to divide-and-conquer routines. For example, when proving the sortedness property (6.3) of the `Insertionsort` algorithm of Figure 6.4, we use saturation with lemma (6.22) applied to `insert`. As such, sortedness of `Insertionsort` is established by the following instance of (6.22):

$$\forall x_a, xs_a \,.\, sorted(xs) \rightarrow sorted(insert(x, xs)) \tag{6.24}$$

(ii) ***Combining reductions preserves permutation equivalence.*** Similarly to Section 6.4.2, proving permutation equivalence (6.4) over divide-and-conquer sorting algorithms of Figure 6.2 is established via the following two properties:

- As in **(P1)** for `Quicksort`, we require that *combine* commutes with $filter_{=}$:

$$\forall x_a, xs_a, ys_a \,.\, filter_{=}(x, combine(xs, ys)) = combine(filter_{=}(x, xs), filter_{=}(x, ys)) \tag{6.25}$$

Note that lemma (6.18) for `Quicksort` is an instance of (6.25), as the *append* function of `Quicksort` acts as a *combine* function of Figure 6.2.

• Similarly to **(P2)** for `Quicksort`, we ensure that, by combining (inverse) *reduction* functions, we preserve (6.4). That is,

$$\forall x_a, xs_a.\ filter_{=}(x, xs) = combine(filter_{=}(x, partition_{\circ}(xs)), \\ filter_{=}(x, partition_{\circ^{-1}}(xs))) \tag{6.26}$$

Note that lemma (6.19) for `Quicksort` is an instance of (6.26), as the $filter_{<}$ and $filter_{\geq}$ functions correspond to the (inverse) *partition* functions of Figure 6.2.

To prove the permutation equivalence (6.4) property of `Mergesort`, we use the functions `take` and `drop` as the *partition* functions of lemmas (6.25)–(6.26). Doing so, we embed a natural number n argument (of sort $\mathbb{N}$) in lemmas (6.25)–(6.26), with n controlling how many list elements are *taken* and *dropped*, respectively, in `Mergesort`. As such, the following instances of lemmas (6.25)–(6.26) are adjusted to `Mergesort`:

$$\forall x_a, xs_a, ys_a.\ filter_{=}(x, merge(xs, ys)) = append(filter_{=}(x, xs), \\ filter_{=}(x, ys)) \tag{6.27}$$

and

$$\forall x_a, n_{\mathbb{N}}, xs_a.\ filter_{=}(x, xs) = append(filter_{=}(x, take(n, xs)), \\ filter_{=}(x, drop(n, xs))), \tag{6.28}$$

with lemmas (6.27)–(6.28) being proved via saturation. With these lemmas at hand, the permutation equivalence (6.4) of `Mergesort` is established, similarly to `Quicksort`. Finally, the generality of lemmas (6.25)–(6.26) naturally pays off when proving the permutation equivalence property (6.4) of `Insertionsort`. Here, we only use a simplified instance of (6.25) to prove (6.4) is preserved by the auxiliary function `insert`. That is, we use the following instance of (6.25):

$$\forall x_a, y_a, ys_a.\ filter_{=}(x, \mathsf{cons}(y, ys)) = filter_{=}(x, insert(y, ys)), \tag{6.29}$$

which is automatically derivable by saturation with computation induction (6.5).

We conclude by emphasizing the generality of the lemmas (6.22) and (6.25)–(6.26) for automating inductive reasoning over sorting algorithms in saturation-based first-order theorem proving: functional correctness of `Quicksort`, `Mergesort`, and `Insertionsort` are proved using these lemmas in saturation with induction. Moreover, each of these lemmas is established via saturation with induction. Thus, compositional reasoning in saturation with computation induction enables proving challenging sorting algorithms in a fully automated manner.

6.6 Related Work

While `Quicksort` has been proven correct on multiple occasions, first and foremost in the famous 1971's pen-on-paper proof by Foley and Hoare [FH71], not many have

investigated a fully automated proof of the algorithm. A partially automated proof of `Quicksort`relies on `Dafny` [Lei10], where loop invariants are manually provided [CDEM+16]. While [CDEM+16] claims to prove some of the lemmas/invariants, not all invariants are proved correct (only assumed to be so). Similarly, the `Why3` framework [FP13] has been leveraged to prove sortedness and permutation equivalence of `Mergesort` [Lév14] over parameterized lists and arrays. These proofs also rely on manual proof splitting with the additional overhead of choosing the underlying prover for each subgoal as `Why3` is interfaced with automated first-order and SMT solvers as well as interactive theorem provers.

The work of [WS04] reports on the verification of functional implementations of multiple sorting algorithms with `VeriFun` [WS03]. Specifically, the correctness of the sortedness property of `Quicksort` is established with the help of 13 auxiliary lemmas while also establishing the permutation property of `Mergesort` by comparing the number of elements, thus requiring additional axiomatization on integer addition. In contrast, our proofs involve less auxiliary lemmas, avoid the overhead of arithmetic reasoning through our formalization of the permutation property over set equivalence and prove functional implementations with arbitrary sorts permitting a linear order.

The work of [SH20] establishes the correctness of permutation equivalence for multiple sorting algorithms based on separation logic through inductive lemmas. However, [SH20] does not address the correctness proofs of the sortedness property. Contrarily, we automate the correctness proofs of sorting algorithms, using compositional first-order reasoning in the theory of parameterized lists.

Verifying functional correctness of sorting routines has also been explored in the abstract interpretation and model-checking communities, by investigating array-manipulating programs [GMT08, JM07]. In [GMT08], the authors automatically generate loop invariants for standard sorting algorithms of arrays of fixed length; the framework is, however, restricted solely to inner loops and does not handle recursive functions. Further, in [JM07] a priori given invariants/interpolants are used in the verification process. Unlike these techniques, we do not rely on a user-provided inductive invariant, nor are we restricted to inner loops.

There are naturally many examples of proofs of sorting algorithms using interactive theorem proving (ITP), see e.g. [JZ17, Lam20]. The work of [JZ17] establishes correctness of insertion sort. Similarly, the setting of [Lam20] proves variations of `Introsort` and `Pdqsort` – both using `Isabelle/HOL` [WPN08]. However, ITP relies on user guidance to provide induction schemes, a burden that we eliminate in our approach.

Further, Beckert et al. [BSSU17] verified a real-world implementation of `Quicksort`, namely Java's inbuilt dual pivot `Quicksort` class, with the semi-automatic `KeY` prover [ABB+05]. The `KeY` system can be understood as an interactive theorem prover for object-oriented programming languages, most notably Java, offering some automation through the integration of multiple fully automated solvers based on SMT and bounded model checking. Java's `Dual Pivot Quicksort` class comprises multiple different sorting routines including `Mergesort` whose choice to sort arrays of Java's primitive data types depends on multiple factors such as the data type of array elements, array

size and structure. Additionally, the KeY prover has also been leveraged to analyze industrial implementations of RadixSort and CountingSort [dGdBR16]. By relying on inductive method annotations such as loop invariants or method contracts and the user to guide the proof rule application during the verification process, the work faces similar limitations as the ones using Dafny, Why3 or ITP. While we manually split our proofs into multiple steps, our lemmas are proved automatically thanks to saturation-based theorem proving with structural/computation induction. As such, we do not require guidance on rule application or inductive annotations. However, it's important to note that the goal of these works is very different from ours. Instead of proving a real-world implementation in a machine-assisted manner, our purpose is to push the boundaries of automating proofs that require induction even further.
When it comes to the landscape of automated saturation-based reasoning, we are not aware of other techniques enabling the fully automated verification of such sorting routines, with or without compositional reasoning.

CHAPTER 7

Tooling and Implementation

This chapter extends our paper

> Pamina Georgiou, Bernhard Gleiss, Ahmed Bhayat, Michael Rawson, Laura Kovács, and Giles Reger. The Rapid Software Verification Framework. In *Proceedings of the 22nd International Conference on Formal Methods in Computer-Aided Design (FMCAD 2022)*, pages 255-260. IEEE, 2022.

Our approaches are based on the Rapid verification framework and/or the first-order theorem prover Vampire. We highlight implementation details for reasoning in trace logic with the Rapid verification framework and establish extensions made to the automated theorem prover Vampireto automate inductive reasoning for verification purposes.

7.1 The Rapid Verification Framework

We now present the Rapid framework for automatic software verification by applying first-order reasoning in trace logic $\mathcal{L}$. Rapid establishes partial correctness of programs with loops and arrays by inferring invariants necessary to prove program correctness using a saturation-based automated theorem prover. Rapid can heuristically generate trace lemmas (Chapter 3), common program properties that guide inductive invariant reasoning. Alternatively, Rapid can exploit nascent support for induction in modern provers to fully automate inductive reasoning without the use of trace lemmas (Chapter 4). In addition, Rapid can be used as an invariant generation engine (Chapter 5), supplying other verification tools with quantified loop invariants necessary for proving partial program correctness.
In this chapter, we present what Rapid can do, sketch its design (Section 7.1.1), and describes its main components and implementation aspects (Sections 7.1.2–7.2.3). Experimental evaluation using the SV-Comp benchmark [Bey21] showing Rapid's efficacy in verification is explored in Chapter 8.

Given a program loop annotated with pre/post-conditions, Rapid offers two modes for proving partial program correctness. In the first, Rapid relies on so-called *trace lemmas*, a priori identified inductive properties that are automatically instantiated for a given program as described in Chapter 3. In the second, Rapid delegates inductive reasoning to the underlying first-order theorem prover [HKV21, RV19], without instantiating trace lemmas as given in Chapter 4. In either mode, the automated theorem prover used by Rapid is Vampire [KV13]. Rapid can also synthesize quantified invariants from program semantics, complementing other invariant-generation methods as discussed in Chapter 5.

Related Work. A prominent line of research in verifying programs with unbounded data structures can use model checking for invariant synthesis. Tools like Spacer/Quic3 [KCSG20, GSV18], SeaHorn [GKKN15] or FreqHorn [FPMG19] are based on constrained horn clauses (CHC) and use either fixed-point calculation or sampling/enumerating invariants until a given safety assertion can be proved. These approaches use SMT solvers to check validity of invariants and are mostly limited to quantifier-free or universally-quantified invariants. Recurrence solving and data-structure-specific tactics can be used to infer and prove quantified program properties [RL18]. Diffy [CGU21] and Vajra [CGU20] derive relational invariants of two mutations of a program such that inductive properties can be enforced over the entire program, without invariants for each individual loop. In contrast to these works, Rapid is not limited to universal or quantifier-free safety assertions but can prove and infer loop properties, such as invariants, in full first-order theories, possibly with alternations of quantifiers (see example in Figure 3.2).

7.1.1 System Overview of Rapid

The Rapid framework consists of approximately 10,000 lines of C++ [1]. Figure 7.1 summarizes the Rapid workflow. Inputs to Rapid are programs P written in $\mathcal{W}$ along with properties F expressed in $\mathcal{L}$. Rapid uses smt-lib syntax [BFT17] to encode properties in $\mathcal{L}$. *Preprocessing* in Rapid applies program transformations for common loop-altering programming constructs, such as `break`, `continue` and early-`return` statements, as well as *timepoint inlining* to obtain a simplified program P' from P (see Section 7.1.2).
Next, Rapid performs *inductive verification* (see Section 7.2) by generating the axiomatic semantics $[\![P']\!]$ expressed in $\mathcal{L}$ and instantiating a set $L_1, ..., L_n$ of inductive properties — so-called *trace lemmas* — for the respective program variables of P'. For establishing some property F, Rapid supports two modes of inductive verification: *standard* and *lemmaless* mode. Both modes generate a first-order verification task corresponding to a partial correctness statement: from the semantics $[\![P']\!]$ and from some lemmas $L_1, ..., L_n$, prove F. The difference in both versions relates to the underlying support for automating inductive reasoning while proving F. The *standard* verification mode

[1] available at `https://github.com/vprover/rapid`

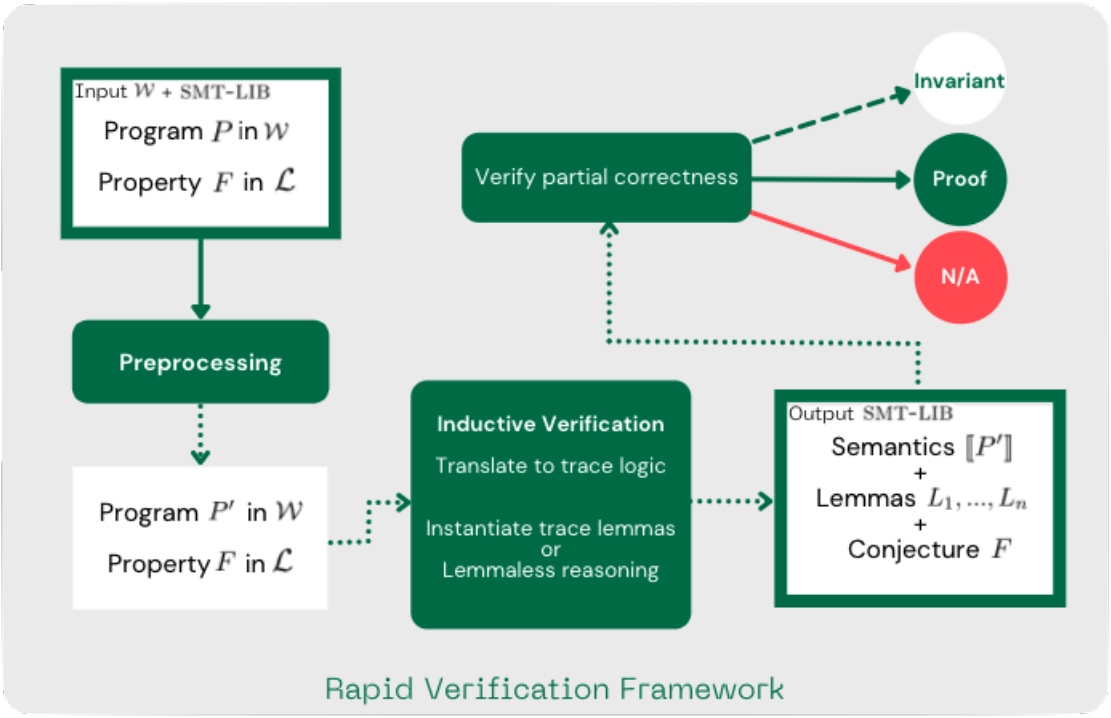

Figure 7.1: Overview of the Rapid verification framework.

equips the verification task with the trace lemmas $L_1, ..., L_n$, providing helpful induction schemes for proving F. The *lemmaless* verification mode uses built-in inductive reasoning and relies less, or not at all, on trace lemmas. In either mode, the verification tasks of Rapid are encoded in the SMT-LIB format. Finally, a third Rapid mode can be used for invariant generation by appropriating symbol elimination for trace logic (see Chapter 5). In this mode, Rapid outputs (quantified) logical consequences of the input program semantics using SMT-LIB syntax; these consequences might represent necessary loop invariants that can further be used by other verification tools.
For proving verification tasks, and thus *verifying partial correctness*, Rapid uses the first-order theorem prover Vampire (see Section 7.3). To this end, we extended the SMT-LIB format to support Rapid-style reasoning and devised Rapid-specific portfolio modes in Vampire, in the spirit of [WL99].

7.1.2 Preprocessing in Rapid

We describe the main translations and optimizations Rapid performs to simplify its verification task.

Program Transformations. We use standard program transformations to translate away `break`, `continue` and `return` statements. For these, Rapid introduces fresh

```
1  while(i < alength) {
2    if (a[i] == x) {
3      break;
4    }
5    i = i + 1;
6  }
7
```

```
1  Bool break = false;
2  while(i < alength && !break )
       {
3    if (a[i] == x) {
4      break = true;
5    }
6    if (!break) {
7      i = i + 1;
8    }
9  }
10
```

Figure 7.2: Loop transformation for `break`-statement.

Boolean program variables indicating whether a statement has been executed. The program is adjusted accordingly: `return` statements end program execution; `break` statements invalidate the first enclosing loop condition; and for `continue` the remaining code of the first enclosing loop body is not executed.
For `break` and `return` we must ensure that the relevant enclosing loop conditions additionally check that the statement has not been executed. We note that `return` statements are defined to work as *early returns*, which implies termination of the entire program. A `break` statement only requires modifying the first enclosing loop condition, but an early-`return` results in execution stopping altogether. Therefore, all enclosing loops and any code following a block containing a `return`-statement will also be guarded and only executed if `return` is not yet reached. For `continue` it is only required that the remaining code of the first enclosing loop body is only executed when `continue` was not reached, and loop execution may continue to the next iteration.

Example 14. We exemplify used program transformations in Rapid with three examples. Figure 7.2 shows a standard transformation for a `break` statement. Figure 7.3 illustrates such a transformation for `continue`, while Figure 7.4 demonstrates the transformation of an early `return` statement breaking program execution.

Timepoint Inlining. Rapid uses SSA-style inlining [BC94, App98, App04] for timepoints to simplify axiomatic program semantics and trace lemmas of a verification task. Specifically, Rapid caches (i) for each integer variable the current program expression assigned to it, and (ii) for each integer-array variable the last timepoint where it was assigned. Cached values are used during traversal of the program tree to simplify later program expressions by removing unnecessary equalities. Thus we avoid defining irrelevant equalities of program variable values over unused timepoints, and only reference timepoints relevant to the property. This option can be toggled with a flag `-inlineSemantics on/off` and is on by default. We illustrate this on two examples:

```
1  while (i < alength) {
2    if (a[i] < 0) {
3      i = i + 1;
4      continue;
5    }
6    i = i + 1;
7    sum = sum + a[i];
8  }
9
```

```
1  Bool continue = false;
2  while (i < alength) {
3    if (a[i] < 0) {
4      i = i + 1;
5      continue = true;
6    }
7    if(!continue){
8      i = i + 1;
9      sum = sum + a[i];
10   }
11 }
12
```

Figure 7.3: Loop transformation for `continue`-statement.

```
1  while (i < alength) {
2    if (a[i] == x) {
3      return i;
4    }
5    i = i + 1;
6  }
7  found = 1;
8
```

```
1  Bool return = false;
2  while (i < alength && !return
       ) {
3    if (a[i] == x) {
4      return = true;
5    }
6    if (!return) {
7      i = i + 1;
8    }
9  }
10 if (!return) {
11   found = 1;
12 }
13
```

Figure 7.4: Loop transformation for `return`-statement.

Example 15 (Inlining assignments.)**.** The effect of inlined semantics can be observed when we encounter block assignments to integer variables: we can skip assignments and use the last assigned expression directly in any reference to the original program variable. Consider the partial program in Figure 7.5a. Our axiomatic semantics in trace logic [GGK20a] would result in

$$\begin{aligned} a(l_2) = a(l_1) + 2 \quad &\wedge \quad b(l_2) = b(l_1) \quad \wedge \\ c(l_2) = c(l_1) \quad &\wedge \quad a(l_3) = a(l_2) \quad \wedge \\ b(l_3) = 3 \quad &\wedge \quad c(l_3) = c(l_2) \quad \wedge \\ a(l_{end}) = a(l_3) \quad &\wedge \quad b(l_{end}) = b(l_3) \quad \wedge \\ c(l_{end}) = a(l_3) + b(l_3) & \end{aligned}$$

```
1  a = a + 2;
2  b = 3;
3  c = a + b;
4
5  assert(a(end) < c(end))
6
```

(a) Partial program with constant assignment.

```
1  if (x > 1) {
2    skip;
3  } else {
4    x = 0;
5  }
6  while (y > 0) {
7    y = y - 1;
8  }
9  assert(x(end) ≥ 0)
10
```

(b) Partial program with branching.

Figure 7.5: Examples for Value Inlining

whereas the inlined version of semantics is drastically shorter:

$$a(l_{end}) = a(l_1) + 2 \qquad \wedge \qquad c(l_{end}) = (a(l_1) + 2) + 3.$$

In contrast to the extended semantics that define all program variables for each timepoint, the inlined version only considers the values of referenced program variables at the timepoint of their last assignment. Thus, when `c` is defined, Rapid directly references the (symbolic) values assigned to `a` and `b`. While `b` is not defined at all, note that `a` *is* defined as $a(l_{end})$ is referenced in the conjecture. Furthermore, the inlined semantics only make use of two timepoints, l_1, and l_{end}, as the remaining timepoints are irrelevant to the conjecture.

Example 16 (Inlining with branching.)**.** Figure 7.5b shows another program that benefits from inlining equalities, as well as only considering timepoints relevant to the conjecture. The original semantics defines program variables `x` and `y` for all program locations: l_1, l_2, l_3, l_4, $l_6(it)$, $l_6(nl_6)$, l_{end}, for some iteration it and final iteration nl_6. While the program contains two variables `x` and `y`, only `x` is used in the property we want to prove. Since no assignments to `x` contain any references to `y`, the loop semantics do not interfere with `x`, so Rapid produces

$$\begin{aligned} &x(l_3) < 1 \rightarrow x(l_6(0)) = x(l_3) && \wedge \\ &x(l_3) \geq 1 \rightarrow x(l_6(0)) = 0 && \wedge \\ &\quad x(l_{end}) = x(l_6(0)) \end{aligned}$$

where the semantics of the loop defining `y` are omitted by the tool. Note that all timepoints of the if-then-else statements are flattened into the timepoint at the beginning of the loop at l_6 in iteration 0. The axiomatic semantics thus reduce to three conjuncts defining the value of `x` throughout the execution. However, `x` is not defined in any loop iteration other than the first as they are irrelevant to the property.

User-defined input. RAPID is fully automated. However, it may still benefit from manually-defined invariants to support the prover. Users can therefore extend the input to RAPID with first-order axioms written in the SMT-LIB format with our proprietary idiom `(axiom ())`.

7.2 Verification Modes

As mentioned above, RAPID implements two verification modes; in the default *standard* mode, RAPID uses trace lemmas to prove inductive properties of programs. In its *lemmaless* mode RAPID relies on built-in induction support in saturation-based first-order theorem proving. In this section we elaborate on both modes further.

7.2.1 Standard Verification Mode: Reasoning with Trace Lemmas

RAPID's *standard* mode relies on trace lemma reasoning to automate inductive reasoning. Trace lemmas are sound formulas that are: (i) derived from bounded induction over loop iterations; (ii) represent common inductive program properties for a set of similar input programs; and (iii) are automatically instantiated for all relevant program variables of a specific input program during its translation to trace logic; see Chapter 3.
In all of our experiments from Section 8, including the examples from Figure 3.1 and Figure 3.2, we only instantiate three generic inductive trace lemmas to establish partial correctness. One such trace lemma asserts, for example, that a program variable is not mutated after a certain execution timepoint as discussed in Chapter 3.

Multitrace Generalization. RAPID can also be used to prove k-safety properties over k traces, useful for security-related hyperproperties such as non-interference and sensitivity [BEG+19]. For such problems it is sufficient to extend program variables to functions over time and trace, such that program variables are represented as $(\mathbb{L} \times \mathbb{T} \mapsto \mathbb{I})$. Program locations, and hence timepoints, are similarly parameterized by an argument of sort $\mathbb{T}$ to denote the same timepoint in different executions.

7.2.2 Lemmaless Verification Mode

When in *lemmaless* mode RAPID does not add any trace lemma to its verification task but relies on first-order theorem proving to derive inductive loop properties. An extended version of SMT-LIB (see Section 7.3) is used to provide the underlying prover with additional information to guide the search for necessary inductive schemes, such as likely symbols for induction. We further equip saturation-based theorem proving with two new inference rules that enable induction on such terms; see [BGE+22] for details. *Multi-clause goal induction* takes a formula derived from a safety assertion that contains a final loop counter, that is a symbol denoting last loop iterations, and inserts an instance of the induction schema for natural numbers with the negation of this formula as its conclusion into the proof search space. For example, consider the

formula $x(l_5(nl_5)) < 0$. Multi-clause goal induction introduces the induction hypothesis $x(l_5(0)) \geq 0 \ \wedge \ \forall it_{\mathbb{N}}.\ (it < nl_5 \ \wedge \ x(l_5(it)) \geq 0) \rightarrow x(l_5(s(it))) \geq 0 \ \rightarrow \ x(l_5(nl_5)) \geq 0$. If the base and step cases can be discharged, a contradiction can be easily produced from the conclusion and original clause.
Array mapping induction also introduces an instance of the induction schema to the search space, but is not based on formulas derived from the goal. Instead, this rule uses clauses derived from program semantics to generate a suitable conclusion for the induction hypothesis. For implementation details of these rules in the underlying saturation-based theorem prover, we refer to Section 7.4.1.

Extensions to Rapid. Rapid takes as an input a $\mathcal{W}$ program along with a property expressed in $\mathcal{L}$. It outputs the semantics of the program expressed in $\mathcal{L}$ using SMT-LIB syntax along with the property to be proven. For our "lemmaless induction" framework, we have extended Rapid as follows. Firstly, we prevent the output of all trace lemmas other than trace lemma C (Section 4.2). We added custom extensions to the SMT-LIB language to identify trace logic symbols, such as loop iteration symbols, program variables, within the Rapid encodings. This way, trace logic symbols to be used for induction inferences are easily identified and can also be used for various proving heuristics. We refer to this version (available online[2]) as $\text{Rapid}_{\text{lemmaless}}$. To run use the command `./rapid -dir OUTPUT_DIR -outputTraceLemmas off BENCHMARK`.

7.2.3 Invariant Generation Mode

Rapid can also be used as an invariant generation engine, synthesizing first-order invariants using the Vampire theorem prover. Rather than performing a proof of partial correctness for some specific safety assertion, Rapid can be used to generate consequences purely from program semantics. Some of these consequences may be loop invariants. To do so, we use a special mode of Vampire to derive logical consequences of the semantics produced by Rapid. However, there is no guarantee that relevant consequences will be derived quickly or at all, as there may be an infinite number of consequences. Therefore, some heuristics must be applied to guide consequence finding for invariant synthesis.
The *symbol elimination* approach of [KV09] defined some set of program symbols undesirable, and only reports consequences that have *eliminated* such symbols from their predecessors. In Rapid, we adjust symbol elimination for deriving invariants with trace logic using Vampire's mode for symbol elimination (Option `-symbol_elimination on`). These invariants may contain quantifier alternations, and some conjunction of them may well be enough to help other verification tools show some property. Specifically, we eliminate colored trace logic-specific symbols and obtain (quantified) formulas as conclusions containing only transparent symbols that are either target symbols, constant program variables (no timepoints), or theory symbols as described in Chapter 5.

[2] See commit `285e54b7e` of `https://github.com/vprover/rapid/tree/ahmed-induction-support`.

When RAPID is in *invariant generation* mode, the encoding of the problem is optimized for invariant generation. We do not include the conjecture in the main problem encoding, but print it to a separate file. This allows for generated consequences to be checked against it to determine a conjunction of consequences strong enough to prove it. Further, we limit trace lemmas to more specific versions of the bounded induction scheme. We also remove RAPID-specific symbols such as lemma literals (Option `-inlineLemmas on`). This disables splitting trace lemmas by introducing lemma literals, instead we the basic form of trace lemmas:

$$P_1 \wedge ... \wedge P_n \rightarrow Conclusion_L$$

Invariant generation mode in RAPID is enabled with the flag `-invariantGeneration on`.

7.3 Verifying Partial Correctness in Rapid

For proving the verification tasks of Section 7.2, and thus verifying partial program correctness, RAPID relies on saturation-based first-order theorem proving. To this end, each verification mode of RAPID uses the VAMPIRE prover, for which we implemented the following, RAPID-specific adjustments.

7.3.1 Extending smt-lib

Each verification task of RAPID is expressed in extensions of SMT-LIB, allowing us to treat some terms and definitions in a special way during proof search:

(i) `declare-nat`: The VAMPIRE prover has been extended with an axiomatization of the natural numbers as a term algebra, especially for RAPID-style verification purposes. We use the command `(declare-nat Nat zero s p Sub)` to declare the sort `Nat`, with constructors `zero` and successor `s`, predecessor `p` and ordering relation `Sub`.

(ii) `declare-lemma-predicate`: Our trace lemmas are usually of the form $(P_1 \wedge ... \wedge P_n) \rightarrow Conclusion_L$ for some trace lemma L with premises $P_1 \wedge ... \wedge P_n$. In terms of reasoning, it makes sense for the prover to derive the premises of such a lemma before using its conclusion to derive more facts, as we have many automatically instantiated lemmas of which we can only prove the premises of some from the semantics. To enforce this, we adapt literal selection such that inferences from premises are preferred over inferences from conclusions. Lemmas are split into two clauses $(P_1 \wedge ... \wedge P_n) \rightarrow Premise_L$ and $Premise_L \rightarrow Conclusion_L$, where $Premise_L$ is declared as a *lemma literal*. We ensure our literal selection function selects either a negative lemma literal[3] if available, or a positive lemma literal only in combination with another literal, requiring the prover to resolve premises before using the conclusion.

[3]Note that lemma literals become negative in the premise definition after CNF-transformation.

The *lemmaless* mode of Rapid introduces the following additional declarations to SMT-LIB:

(i) `declare-const-var`: This declaration is used to assign symbols representing constant program variables a large weight in the prover's term ordering, allowing constant variables to be rewritten to non-constant expressions.

(ii) `declare-program-var`: Rapid declares mutable program variables with this keyword for the prover to be able to differentiate between constant and other program variables.

(iii) `declare-timepoint`: This declaration indicates to the prover that symbol represents a timepoint to distinguish from program variables, guiding Vampire to apply induction upon timepoints.

(iv) `declare-final-loop-count`: This keyword declares a symbol as a final loop count symbol, thus eligible for induction.

7.3.2 Portfolio Modes

We further developed a collection of Rapid-specific proof options in Vampire, using for example extensions of theory split queues [GS20] and equality-based rewritings [GKR20]. Such options have been distilled into a Rapid portfolio schedule that can be run with `--mode portfolio -sched rapid`. Moreover, the multi-clause goal induction rule and the array mapping induction inference of Rapid have been compiled to a separate portfolio mode, accessed via `--mode portfolio -sched induction_rapid`.

7.4 The Vampire Theorem Prover

All of our work is based on the theorem prover Vampire. Beyond making changes to the parser according to input formats in Rapid or for recursion induction over parameterized lists, we here give some details on our implementation of the various forms of built-in induction used in this work.

7.4.1 Lemmaless Reasoning in Vampire

We implemented the `MCGLoopInd` inference rule and a slightly simplified version of the `AMLoopInd` rule in a new branch of Vampire[4]. The new induction support is diverging from previous induction work in Vampire and controlled by a separate set of options. The main issue with the induction inferences `MCGLoopInd` and `AMLoopInd` is their explosiveness which can cause proof search to diverge. We have, therefore, introduced various heuristics in the implementation to try and control them. For `MCGLoopInd` we not only necessitate that the premises are derived from the conjecture, but that their derivation length from the conjecture is below a certain distance controlled by an option. The premises must be unit clauses unless another option `multi_literal_ clauses` is toggled on. The option `induct_all_loop_counts` allows `MCGLoopInd` induction

[4]See commit `4a0f319f` of `https://github.com/vprover/vampire/tree/ahmed-rapid`.

to take place on all loop counter terms, not just final loop iterators. In order for the `MCGLoopInd` and `AMLoopInd` inferences to be applicable, we need to rewrite terms not containing final loop counters to terms that do. However, rewriting in Vampire is based on superposition, which is parameterised by a term order preventing smaller terms to be rewritten into larger ones. In this case, the term order may work against us and prevent such rewrites from happening. We implemented a number of heuristics to handle this problem. One such heuristic is to give terms representing constant program variables a large weight in the ordering. Then, equations such as $alength \simeq i(tp_{\mathtt{w}}(nl_{\mathtt{w}}))$ will be oriented left to right as desired. We combined these options with others to form a portfolio of strategies[5] that contains 13 strategies each of which runs in under 10s.

7.4.2 Recursion Induction in Vampire

Beyond portfolio modes for Rapid-style verification, we extend Vampire with induction inference rules to handle computation induction over parameterized lists (see Chapter 6). Our work on saturation with induction in the first-order theory of parameterized lists (Chapter 6) is implemented in the first-order prover Vampire [KV13]. In support of parameterization, we extended the SMT-LIB parser of Vampire to support parametric data types from SMT-LIB [BFT16] – version 2.6. In particular, using the `par` keyword, our parser interprets `(par (`$\mathtt{a}_1$ `...` $\mathtt{a}_n$`) ...)` similar to universally quantified blocks where each variable $\mathtt{a}_i$ is a type parameter. That is, parametric functions are specified via `(declare-fun f (par ...))` and `(define-fun[-rec] f (par ...))`.
Appropriating a generic saturation strategy, we adjust the simplification orderings (LPO) for efficient equality reasoning/rewrites to our setting. For example, the precedence of function *quicksort* is higher than of symbols nil, cons, *append*, $filter_<$ and $filter_\geq$, ensuring that *quicksort* function terms are expanded to their functional definitions.
We further apply recent results of encompassment demodulation [DK22] to improve equality reasoning within saturation (`-drc encompass`). We use induction on data types (`-ind struct`), including complex data type terms (`-indoct on`).

[5] `--mode portfolio --schedule rapid_induction [benchmark.smt2]`

CHAPTER 8

Experiments and Evaluation

In this chapter, we report on our experimental evaluations performed on Rapid's reasoning capabilities (Section 8.1), as well as on recursion induction for sorting algorithms (Section 8.2).

8.1 Rapid Experimental Results

8.1.1 Benchmarks

For our experimental evaluation of Rapid, we use a total of 111 examples whose verification involved proving safety assertions of different logical complexity (quantifier-free, only universally/existentially quantified, and with quantifier alternations). Our benchmarks are divided into four groups, as indicated in Table 8.1: (i) the first 13 problems have quantifier-free proof obligations; (ii) the majority of benchmarks, in total 68 examples, contain universally quantified safety assertions; (iii) 7 problems come with the task of verifying existentially quantified assertions; (iv) and the last 23 programs contain assertions with quantifier alternation.
The examples from (i)-(ii), a total of 81 programs, come from the array verification benchmarks of the SV-Comp [Bey12] repository [1], with most of these examples originating from [DDA10, GSV18].[2] These examples correspond to the set of those SV-Comp benchmarks which use the fragment of C supported by $\mathcal{W}$ in Rapid; specifically, when selecting examples (i)-(ii) from SV-Comp, we omitted examples containing pointers or memory management. In general SV-Comp benchmarks are bounded to a certain array size N. By contrast, we treat arrays as unbounded in Rapid and reason using arbitrary but fixed symbolic array lengths. All SV-Comp input problems from (i)-(ii) are thus

[1] https://github.com/sosy-lab/sv-benchmarks

[2] In order to reproduce the results reported in Table 8.1, instructions are provided at https://github.com/vprover/vampire_publications/tree/master/experimental_data/CICM-2022-RAPID-INDUCTION

adapted to our input format, and pre-/postcondition pairs are translated to trace logic formulas. Further, benchmarks (iii)-(iv) are new examples crafted by us, in total 30 new problems. They specifically contain existential and alternating quantification in safety assertions to highlight Rapid's capabilities in contrast to SMT-based verification approaches.

8.1.2 Experimental Setting

We used both versions of Rapid in our experiments - the standard and the lemmaless mode as described in Section 7.2. First, **(1)** $\text{Rapid}_{\text{lemmaless}}$ denotes our Rapid approach, using lemmaless induction `MCGLoopInd` and `AMLoopInd` in Vampire (see Chapter 4). Further, **(2)** $\text{Rapid}_{\text{std}}$ uses trace lemmas for inductive reasoning, as described in Chapter 3. We also compared $\text{Rapid}_{\text{lemmaless}}$ with other verification tools. In particular, we considered **(3)** SeaHorn and **(4)** Vajra (and its extension Diffy that produced for us exactly the same results as Vajra). SeaHorn converts the program into a constrained horn clause (CHC) problem and uses the SMT solver Z3 for solving. Vajra and Diffy implement inductive reasoning and recurrence solving over loop counters; in the background, they also use Z3. We summarize our findings below.

8.1.3 Results

Table 8.1 shows that $\text{Rapid}_{\text{lemmaless}}$ is superior to $\text{Rapid}_{\text{std}}$ for the given benchmark set, as it solves a total of 93 problems, while $\text{Rapid}_{\text{std}}$ proves 78 assertions correct. Particularly, $\text{Rapid}_{\text{lemmaless}}$ can solve benchmark `merge_interleave_2` corresponding to example 4.1, and other challenging problems such as `find_max_local_1` also containing quantifier alternations, while maintaining most results proven by $\text{Rapid}_{\text{std}}$. $\text{Rapid}_{\text{lemmaless}}$ could establish correctness of a total of 18 problems that $\text{Rapid}_{\text{std}}$ could not solve. It is, thus, interesting to look into which problems $\text{Rapid}_{\text{lemmaless}}$ solves: many of the newly solved safety assertions are structurally very close to the loop invariants needed to prove them. This is where multi-clause goal-oriented induction `MCGoalInd` makes the biggest impact. For instance, this allows $\text{Rapid}_{\text{lemmaless}}$ to prove the partial correctness of `find_max_ from_second_0` and `find_max_from_second_1`.
On the other hand, $\text{Rapid}_{\text{lemmaless}}$ also lost five challenging benchmarks that were previously solved by $\text{Rapid}_{\text{std}}$, such as `swap_0` and `partition_5`. This could be for two reasons: (1) the strategies in the induction schedule of $\text{Rapid}_{\text{lemmaless}}$ are too restrictive for such benchmarks, or (2) the step case of the induction axiom introduced by our two rules are too difficult for Vampire to prove. That is, the prover cannot derive the conclusion which might prove the conjecture. Strengthening lemmaless induction with additional trace lemmas from $\text{Rapid}_{\text{std}}$ is an interesting line of further work.

Comparing with other tools. Both, SeaHorn and Vajra/Diffy require C code as input, whereas Rapid uses its own syntax. We translated our benchmarks to C code expressing the same problem. However, a direct comparison of Rapid with most other verifiers requiring standard C code as an input is not possible as we consider slightly

Table 8.1: Rapid Experimental Results in Detail.

Benchmark	(1)	(2)	(3)	(4)
atleast_one_iteration_0	✓	✓	✓	✓
atleast_one_iteration_1	✓	✓	✓	✓
count_down	✓	-	-	-
eq	✓	-	✓	-
find_sentinel	✓	✓	-	-
find1_0	✓	✓	✓	-
find1_1	✓	✓	✓	-
find2_0	✓	✓	✓	-
find2_1	✓	✓	✓	-
indexn_is_arraylength_0	✓	✓	✓	-
indexn_is_arraylength_1	✓	✓	✓	-
set_to_one	✓	✓	✓	✓
str_cpy_3	✓	✓	✓	-
add_and_subtract	✓	-	-	✓
both_or_none	✓	✓	-	✓
check_equal_set_flag_1	✓	✓	-	✓
collect_indices_eq_val_0	✓	✓	-	✓
collect_indices_eq_val_1	✓	✓	-	-
copy	✓	✓	-	✓
copy_absolute_0	✓	✓	-	✓
copy_absolute_1	✓	✓	-	✓
copy_and_add	✓	-	-	✓
copy_nonzero_0	✓	✓	-	✓
copy_partial	✓	✓	-	✓
copy_positive_0	✓	✓	-	✓
copy_two_indices	✓	✓	-	-
find_max_0	✓	✓	-	✓
find_max_2	✓	✓	-	✓
find_max_from_second_0	✓	-	-	✓
find_max_local_2	-	-	-	-
find_max_up_to_0	-	-	-	-
find_max_up_to_2	-	-	-	-
find_min_0	✓	✓	-	✓
find_min_2	✓	✓	-	-
find_min_local_2	-	-	-	-
find_min_up_to_0	-	-	-	-
find_min_up_to_2	-	-	-	-
find1_4	-	✓	-	-
find2_4	✓	✓	-	-
in_place_max	✓	✓	-	✓
inc_by_one_0	✓	✓	-	✓
inc_by_one_1	✓	✓	-	✓
inc_by_one_harder_0	✓	✓	-	✓
inc_by_one_harder_1	✓	✓	-	✓
init	✓	✓	-	-
init_conditionally_0	✓	✓	-	-
init_conditionally_1	✓	✓	-	✓
init_non_constant_0	✓	✓	-	-
init_non_constant_1	✓	✓	-	✓
init_non_constant_2	✓	✓	-	✓
init_non_constant_3	✓	✓	-	✓
init_non_constant_easy_0	✓	✓	-	-
init_non_constant_easy_1	✓	✓	-	✓
init_non_constant_easy_2	✓	✓	-	✓
init_non_constant_easy_3	✓	✓	-	✓
init_partial	✓	✓	-	✓

Benchmark	(1)	(2)	(3)	(4)
init_prev_plus_one_0	✓	✓	-	-
init_prev_plus_one_1	✓	✓	-	-
init_prev_plus_one_alt_0	✓	✓	-	-
init_prev_plus_one_alt_1	✓	✓	-	-
insertion_sort	-	-	-	-
max_prop_0	✓	✓	-	✓
max_prop_1	✓	✓	-	✓
merge_interleave_0	✓	-	-	✓
merge_interleave_1	✓	-	-	✓
min_prop_0	✓	✓	-	✓
min_prop_1	✓	✓	-	✓
partition_0	✓	✓	-	✓
partition_1	✓	✓	-	✓
push_back	✓	✓	-	✓
reverse	✓	✓	-	-
rewnifrev	✓	-	-	✓
rewrev	✓	-	-	✓
skipped	✓	-	-	✓
str_cpy_0	✓	✓	-	-
str_cpy_1	✓	✓	-	-
str_cpy_2	✓	✓	-	-
swap_0	-	✓	✓	✓
swap_1	-	✓	✓	✓
vector_addition	✓	✓	-	✓
vector_subtraction	✓	✓	-	✓
check_equal_set_flag_0	✓	✓	-	-
find_max_1	-	-	-	-
find_max_from_second_1	✓	-	-	-
find1_2	✓	✓	-	-
find1_3	✓	✓	-	-
find2_2	✓	✓	-	-
find2_3	✓	✓	-	-
collect_indices_eq_val_2	-	✓	-	-
collect_indices_eq_val_3	✓	-	-	-
copy_nonzero_1	✓	✓	-	-
copy_positive_1	✓	✓	-	-
find_max_local_0	-	-	-	-
find_max_local_1	✓	-	-	-
find_max_up_to_1	-	-	-	-
find_min_1	-	-	-	-
find_min_local_0	-	-	-	-
find_min_local_1	✓	-	-	-
find_min_up_to_1	-	-	-	-
merge_interleave_2	✓	-	-	-
partition_2	✓	✓	-	-
partition_3	✓	✓	-	-
partition_4	-	-	-	-
partition_5	-	✓	-	-
partition_6	-	-	-	-
partition-harder_0	✓	✓	-	-
partition-harder_1	✓	✓	-	-
partition-harder_2	✓	-	-	-
partition-harder_3	✓	-	-	-
partition-harder_4	✓	-	-	-
str_len	✓	✓	-	-
Total solved	**93**	**78**	**13**	**47**

Table 8.2: Rapid Extended Experiment Overview

Total	$\text{Rapid}_{\text{std}}$	$\text{Rapid}_{\text{lemmaless}}$	Diffy	SeaHorn
140	91 (5)	103 (10)	61 (1)	17 (0)

different semantics. In contrast to SeaHorn and Vajra/Diffy, we assume that integers and arrays are unbounded and that all array positions are initialized by arbitrary data. Further, we can read/write at any array position without allocating the accessed memory beforehand.
Apart from semantic differences, Rapid can directly express assertions and assumptions containing quantifiers and put variable contents from different points in time into relation thanks to trace logic $\mathcal{L}$. In order to deal with the latter, we introduced history variables in the code provided to SeaHorn and Vajra/Diffy. Quantification was simulated by non-deterministically assigned variables and/or loops. As a result, SeaHorn verified only 13 examples, whereas Vajra/Diffy could solve 47 of our benchmarks. As Vajra/Diffy restrict their input programs to contain only loops having very specific loop-conditions, several of our benchmarks failed. For example, $i < \mathit{length}$ is permitted, whereas $a[i] \neq 0$ is not. Vajra/Diffy could prove correctness for nearly all the programs satisfying these restrictions. SeaHorn, on the other hand, has problems with the complexity introduced by the arrays. It could solve especially those benchmarks whose correctness do not depend on the arrays' content.

8.1.4 Extended Experiments

We subsequently conducted another round of experiments based on the above benchmark set. However, we extended this set to a total of 140 benchmarks to include most of the `c/ReachSafety-Array` category of the SV-Comp repository, specifically from the `array-examples/*` subcategory[3] as it contains problems suitable for our input language.
Many benchmarks in the original SV-Comp repository are minor variations of each other that differ only in one concrete integer value, for example to increment a program variable by some integer. Instead of copying each such variation for different digits, we abstract such constant values to a single symbolic integer constant such that just one of our benchmark covers numerous cases in the original SV-Comp setup.
We again compare our two Rapid verification modes, indicated by $\text{Rapid}_{\text{std}}$ and $\text{Rapid}_{\text{lemmaless}}$ respectively, against SeaHorn and Diffy. All experiments were run on a cluster with two 2.5GHz 32-core CPUs and one TB RAM with a 60-second timeout.

Results. Table 8.2 summarizes our results, parentheticals indicating uniquely solved problems for each solver/configuration. Note that Diffy produced the same results as its precursor Vajra in this experiment. Of a total of 140 benchmarks, $\text{Rapid}_{\text{std}}$ solves 91 problems, while $\text{Rapid}_{\text{lemmaless}}$ surpasses this number by 12 problems. Particularly,

[3] `https://github.com/sosy-lab/sv-benchmarks/tree/master/c/array-examples`

RAPID$_{\text{lemmaless}}$ could solve more variations with quantifier alternations, as property-driven induction works well for such problems, thus confirming our prior experiment. Again a small number of instances was solved by RAPID$_{\text{std}}$ but not by RAPID$_{\text{lemmaless}}$ within the time limit, indicating that trace lemma reasoning can help to fast-forward proof search. In total, RAPID solves 112 unique benchmarks, whereas SEAHORN and DIFFY could respectively prove 17 and 61 problems (with mostly universally quantified properties).

8.2 Computation Induction and Sorting Experiments

8.2.1 Experimental Evaluation

We evaluated our approach discussed in Chapter 6 on challenging recursive sorting algorithms taken from [NBE+21], namely `Quicksort`, `Mergesort`, and `Insertionsort`. The authors list a wide variety of known algorithms in a functional programming style and outline their type-theoretic proofs based on interactive theorem provers. We show that the functional correctness of these sorting routines can be verified automatically by means of saturation-based theorem proving with induction, as summarized in Table 8.3. We divide our experiments according to the specification of sorting algorithms: the upper part refers to the sortedness property (6.3) while the lower part `PermEq` shows the experiments of all sorting routines w.r.t. permutation equivalence (6.4), together implying the functional correctness of the respective sorting algorithm. Here, the inductive lemmas of Sections 6.4–6.5 are proven in separate saturation runs of VAMPIRE with structural/computation induction; these lemmas are then used as input assumptions to VAMPIRE to prove validity of the respective property.[4]
The benchmarks are categorized as follows. A benchmark category `SA-PR[-L`i`]` indicates that it belongs to proving the property `PR` for sorting algorithm `SA`, where `PR` is one of `S` (sortedness (6.3)) and `PE` (permutation equivalence (6.4)) and `SA` is one of `IS` (`Insertionsort`), `MS` (`Mergesort`) and `QS` (`Quicksort`). Additionally, an optional `Li` indicates that the benchmark corresponds to the i-th lemma for proving the property of the respective sorting algorithm.

Experimental Setup. We prove using saturation-based theorem proving with induction: each benchmark or lemma is tested with a portfolio of solver configurations running a total of five minutes on a machine with an AMD Epyc 7502 chip comprising a 2.5 GHz CPU with 1 TB RAM, of which we use 1 core and 16 GB RAM per benchmark. To identify the successful configuration, we ran VAMPIRE on each benchmark in a portfolio setting with strategies enumerating all combinations of options that we hypothesized to be relevant for solving these problems.

Results. For our experiments, we ran all possible combinations of lemmas to determine the minimal lemma dependency for each benchmark. For example, the sortedness property

[4] Benchmarks and instructions to run the experiments can be found at `https://github.com/mina1604/sorting_wo_sorts`.

Sortedness			
Benchm.	Pr.	T	Required lemmas
IS-S	✓	0.01	{IS-S-L1}
IS-S-L1	✓	8.28	-
MS-S	✓	0.08	∅
MS-S-L1	✓*	0	-
MS-S-L2	✓	0.02	∅
QS-S	✓	0.09	{QS-S-L1, QS-S-L2, QS-S-L3}, {QS-S-L1, QS-S-L3, QS-S-L4}
QS-S-L1	✓	0.27	∅
QS-S-L2	✓	0.04	{QS-S-L4}
QS-S-L3	✓	11.82	{QS-S-L4, QS-S-L5}
QS-S-L4	✓	8.28	{QS-S-L6}
QS-S-L5	✓	0	{QS-S-L7}
QS-S-L6	✓	0.02	∅
QS-S-L7	✓	0.02	∅

PermEq			
Benchm.	Pr.	T	Required lemmas
IS-PE	✓	0.02	{IS-PE-L1}
IS-PE-L1	✓	0.13	∅
MS-PE	✓	0.06	{MS-PE-L1, MS-PE-L2}
MS-PE-L1	✓*	0	-
MS-PE-L2	✓	0.03	∅
MS-PE-L3	✓	0.15	∅
QS-PE	✓	0.5	{QS-PE-L1, QS-PE-L2}
QS-PE-L1	✓	0.05	∅
QS-PE-L2	✓	0.09	∅

Table 8.3: Experimental Evaluation of Computation Induction on Sorting Algorithms.

of Quicksort (QS-S) depends on at least three lemmas (see Section 6.4.1) where two different subsets of lemma combinations enable the proof. The third lemma for this property (QS-S-L_3) depends on two further lemmas namely QS-S-L_4 and QS-S-L_5. The second column Pr. indicates that Vampire solved the benchmark during portfolio mode, by using a minimal subset of needed lemmas given in the fourth column. The third column T shows the running time in seconds of the respective saturation run using the first solving strategy identified during portfolio mode.
In accordance with Table 8.3, Vampire compositionally proves permutation equivalence of Insertionsort and Quicksort and sortedness of Mergesort and Quicksort.

Table 8.4: Structural Induction Applications in Proof Search and Proof.

Benchmark	IndProofSearch	IndProof	Benchmark	IndProofSearch	IndProof
IS-S	4	1	QS-S-L1	510	2
IS-S-L1	339	2	QS-S-L2	9	1
IS-PE	5	1	QS-S-L3	130	2
IS-PE-L1	34	1	QS-S-L4	183	3
MS-S	8	1	QS-S-L5	0	0
MS-S-L2	22	1	QS-S-L6	26	1
MS-PE	14	1	QS-S-L7	16	2
MS-PE-L2	16	1	QS-PE	12	1
MS-PE-L3	136	3	QS-PE-L1	10	1
QS-S	10	2	QS-PE-L2	42	4

Note that sortedness of Mergesort is proven without any lemmas, hence lemma MS-S-L_1 is not needed. Interestingly, while MS-S-L_1 is actually synthesized automatically during saturation of MS-S, it could not be verified by means of portfolio configurations in the solver. The lemmas MS-PE-L_1 for the permutation equivalence of Mergesort and IS-S-L_1 for the sortedness of Insertionsort could be proven separately by more tailored search heuristics in Vampire (hence ✓∗), but our cluster setup failed to consistently prove these with the portfolio setting.
From Section 6.4 it is already evident that the sortedness proof of Quicksort is by far the most complex while Mergesort could be established by following the ideas presented in Section 6.5. It is easily seen that simple algorithms not requiring complex recursive calls such as Insertionsort are efficiently proven by automating computation induction: both sortedness and permutation equivalence only require one automatically established lemma respectively.

8.2.2 Inductive Inferences during Proof Search

For all conjectures and lemmas that were proved in portfolio mode, we summarized the applications of inductive inferences with structural and computation induction schemata in Table 8.4. Specifically, Table 8.4 compares the number of inductive inferences performed during proof search (column IndProofSearch) with the number of used inductive inferences as part of each benchmark's proof (column IndProof). While most safety properties and lemmas required less than 50 inductive inferences, thereby using mostly one or two of them in the proof, some lemma proofs exceeded this by far. Most notably IS-S-L1 and QS-S-L1, Insertionsort's and Quicksort's first lemma respectively, depended on many more inductive inferences until the right axiom was found. Such statistics point to areas where the prover still has room to be finetuned for software verification and quality assurance purposes, here especially towards establishing correctness of functional programs.

CHAPTER 9

Conclusion and Future Work

We will now briefly conclude on our work and raise some potential future lines of work. In this thesis, we addressed the problem of inductive reasoning for software verification with automated first-order theorem provers based on the superposition calculus. To that end, we combined and adapted several techniques of induction.

Inductive Reasoning in Trace Logic. We introduced trace lemma reasoning to automatically prove safety properties over programs containing arrays, integers and loops. Trace logic supports explicit timepoint reasoning to allow arbitrary quantification over loop iterations. We introduced trace lemmas – consequences of bounded induction over timepoints – to automate inductive loop reasoning in trace logic. Generalizing our work to termination analysis and extending our programming language, and its semantics in trace logic, with more complex constructs are interesting tasks for future work.
Moreover, we established lemmaless induction to fully automate the verification of inductive properties of programs with loops over unbounded arrays and integers. We introduced goal-oriented and array mapping induction inferences, triggered by loop counters, in superposition-based theorem proving, We, thus, combined reasoning about programs in first-order logic with induction in saturation. Both these are active areas of investigation, and the combination of the two approaches – trace lemma and lemmaless reasoning – shows great promise when it comes to verifying properties over programs that contain loops.
Our results show that lemmaless induction in trace logic outperforms other state-of-the-art approaches in the area. There are various ways to further develop lemmaless induction in trace logic. On larger benchmarks, particularly those containing multiple loops, our approach struggles. For loops where the required invariant is not connected to the conjecture, we introduced array mapping induction. However, the array mapping induction inference is limited in the form of invariants it can generate. Investigating other methods, such as machine learning for synthesizing loop invariants could be a

line of future work. One option would be to use machine learning to suggest possible invariants based on the terms generated by the prover. Another avenue of investigation is to consider how unhelpful induction lemmas with non-provable base or step cases can be rejected early.

Invariant Generation. We revisit the symbol elimination method to generate and extract invariants from superposition-based proof search. These invariants may contain universal, existential as well as alternating quantification over programs with loops, unbounded arrays and linear integer arithmetic and can, thus, complement other state-of-the-art invariant inference techniques. An interesting line of future work is to combine symbol elimination with inductive reasoning for invariant generation, and use Avatar [Vor14] to guide splitting inductive goals. Modifying Avatar to accommodate symbol elimination would be very useful here, as it allows separately solving base and inductive steps, but it is not yet clear how to achieve this.

Rapid Verification Framework. All of these works culminated in the Rapid verification framework for proving partial correctness of programs containing loops and arrays, and its applications towards efficient inductive reasoning and invariant generation. To this end, we implemented and described different reasoning modes that implement trace lemma and lemmaless verification approaches, as well as an invariant generation method through consequence-finding. Extending Rapid and its axiomatic semantics in trace logic with function calls, and automation thereof, is an interesting task for future work. Moreover, a prior result to this thesis [BEG+19] showed that trace logic and trace lemma reasoning can be extended to hyperproperty verification of two-safety properties. However, this work has not been revisited with built-in induction yet and might benefit from saturation-based automated induction.

Inductive Reasoning for Recursion. Apart from trace logic based reasoning, we presented an integrated formal approach to establish program correctness over recursive programs based on saturation-based theorem proving. We automatically prove recursive sorting algorithms, particularly the `Quicksort` algorithm, by formalizing program semantics in the first-order theory of parameterized lists. Doing so, we expressed the common properties of sortedness and permutation equivalence in an efficient way for first-order theorem proving. By leveraging common structures of divide-and-conquer sorting algorithms, we advocate compositional first-order reasoning with built-in structural and computation induction. Proving further recursive sorting/search algorithms in future work, with improved compositionality, is therefore an interesting challenge to investigate. Another path that is to be explored is using higher-order logic and lambda abstractions for reduction operations as a means to make our general strategy wider applicable.
Our work has two primary implications. Firstly, integrating inductive reasoning in automated theorem proving to prove (sub)goals during interactive theorem proving has the potential to significantly reduce the burden of proof engineers manually demonstrating

proof obligations, as automated theorem proving based on our work can synthesize induction hypotheses to validate such conditions. Secondly, the search for reasonable strategies to automatically split proof obligations on input problems can tremendously enhance the level of automation for proofs that require heavy inductive reasoning. We anticipate that our work will open up new directions in the combination of interactive and automated reasoning, by further reducing the manual effort in proof splitting and improving the applicability of superposition frameworks to a broader variety of recursive algorithms.

List of Figures

List of Tables

Bibliography

[ABB+05] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hähnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, et al. The KeY tool: integrating object oriented design and formal verification. *Software & Systems Modeling*, 4:32–54, 2005.

[ABG+12] Francesco Alberti, Roberto Bruttomesso, Silvio Ghilardi, Silvio Ranise, and Natasha Sharygina. Lazy abstraction with interpolants for arrays. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, pages 46–61. Springer, 2012.

[ACC+20] Mohammad Afzal, Supratik Chakraborty, Avriti Chauhan, Bharti Chimdyalwar, Priyanka Darke, Ashutosh Gupta, Shrawan Kumar, Charles Babu, Divyesh Unadkat, and R Venkatesh. VeriAbs: Verification by abstraction and test generation (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 383–387. Springer, 2020.

[AO16] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[App98] Andrew W Appel. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998.

[App04] Andrew W Appel. *Modern compiler implementation in C*. Cambridge University Press, 2004.

[APS14] Matteo Avalle, Alfredo Pironti, and Riccardo Sisto. Formal verification of security protocol implementations: a survey. *Formal Aspects of Computing*, 26:99–123, 2014.

[BC94] Preston Briggs and Keith D Cooper. Effective partial redundancy elimination. *ACM SIGPLAN Notices*, 29(6):159–170, 1994.

[BC13] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development: Coq'Art: the calculus of inductive constructions*, volume 1 of *Texts in Theoretical Computer Science. An EATCS Series (TTCS)*. Springer, 2013.

[BCD+11] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *International Conference on Computer-Aided Verification (CAV)*, pages 171–177. Springer, 2011.

[BEG+19] Gilles Barthe, Renate Eilers, Pamina Georgiou, Bernhard Gleiss, Laura Kovács, and Matteo Maffei. Verifying relational properties using trace logic. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 170–178. IEEE, 2019.

[Bey12] Dirk Beyer. Competition on software verification: (sv-comp). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 504–524. Springer, 2012.

[Bey21] Dirk Beyer. Software verification: 10th comparative evaluation (SV-COMP 2021). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 401–422. Springer, 2021.

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.

[BFT17] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa, 2017. Available at `www.SMT-LIB.org`.

[BG94] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[BG01] L. Bachmair and H. Ganzinger. Resolution theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 2, pages 19–99. Elsevier Science, 2001.

[BGE+22] Ahmed Bhayat, Pamina Georgiou, Clemens Eisenhofer, Laura Kovács, and Giles Reger. Lemmaless induction in trace logic. In *International Conference on Intelligent Computer Mathematics (CICM)*, pages 191–208. Springer, 2022.

[BGMR15] 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*, pages 24–51. Springer, 2015.

[BM90] Robert S Boyer and J Strother Moore. A theorem prover for a computational logic. In *International Conference on Automated Deduction (CADE)*, pages 1–15. Springer, 1990.

[BMR13] Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. On solving universally quantified horn clauses. In *International Symposium on Static Analysis (SAS)*, pages 105–125. Springer, 2013.

[BSSU17] Bernhard Beckert, Jonas Schiffl, Peter H Schmitt, and Mattias Ulbrich. Proving jdk's dual pivot quicksort correct. In *International Conference on Verified Software. Theories, Tools, and Experiments (VSTTE)*, pages 35–48. Springer, 2017.

[BSVH+93] Alan Bundy, Andrew Stevens, Frank Van Harmelen, Andrew Ireland, and Alan Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62(2):185–253, 1993.

[CCL11] Patrick Cousot, Radhia Cousot, and Francesco Logozzo. A Parametric Segmentation Functor for Fully Automatic and Scalable Array Content Analysis. In *Symposium on Principles of Programming Languages (POPL)*, pages 105–118. ACM, 2011.

[CDEM+16] Razvan Certezeanu, Sophia Drossopoulou, Benjamin Egelund-Muller, K Rustan M Leino, Sinduran Sivarajan, and Mark Wheelhouse. Quicksort revisited: Verifying alternative versions of quicksort. *Theory and Practice of Formal Methods: Essays Dedicated to Frank de Boer on the Occasion of His 60th Birthday*, pages 407–426, 2016.

[CG12] Alessandro Cimatti and Alberto Griggio. Software model checking via IC3. In *International Conference on Computer-Aided Verification (CAV)*, pages 277–293. Springer, 2012.

[CGU20] Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. Verifying array manipulating programs with full-program induction. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 22–39. Springer, 2020.

[CGU21] Supratik Chakraborty, Ashutosh Gupta, and Divyesh Unadkat. Diffy: Inductive Reasoning of Array Programs Using Difference Invariants. In *International Conference on Computer-Aided Verification (CAV)*, pages 911–935. Springer, 2021.

[Chu36] Alonzo Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, 1936.

[CJRS13] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In *International Conference on Automated Deduction (CADE)*, pages 392–406. Springer, 2013.

[Cru15] Simon Cruanes. *Extending superposition with integer arithmetic, structural induction, and beyond*. PhD thesis, École polytechnique, 2015.

[DDA10] Isil Dillig, Thomas Dillig, and Alex Aiken. Fluid updates: Beyond strong vs. weak updates. In *European Symposium on Programming (ESOP)*, pages 246–266. Springer, 2010.

[DDH72] Ole-Johan Dahl, Edsger Wybe Dijkstra, and Charles Antony Richard Hoare. *Structured programming*, volume 8 of *A.P.I.C. Studies in Data Processing*. Academic Press Ltd., 1972.

[dGdBR16] Stijn de Gouw, Frank S de Boer, and Jurriaan Rot. Verification of counting sort and radix sort. *Deductive Software Verification–The KeY Book: From Theory to Practice*, pages 609–618, 2016.

[Dij75] Edsger W Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.

[DK22] André Duarte and Konstantin Korovin. Ground joinability and connectedness in the superposition calculus. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 169–187. Springer, 2022.

[DKW08] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340. Springer, 2008.

[DNS05] David Detlefs, Greg Nelson, and James B Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005.

[DP01] N. Dershowitz and D. A. Plaisted. Rewriting. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 9, pages 535–610. Elsevier Science, 2001.

[EP20] Mnacho Echenim and Nicolas Peltier. Combining induction and saturation-based theorem proving. *Journal of Automated Reasoning*, 64(2):253–294, 2020.

[FB18] Grigory Fedyukovich and Rastislav Bodík. Accelerating syntax-guided invariant synthesis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 251–269. Springer, 2018.

[FH71] Michael Foley and Charles Antony Richard Hoare. Proof of a recursive program: Quicksort. *The Computer Journal*, 14(4):391–395, 1971.

[FJS04] Cormac Flanagan, Rajeev Joshi, and James B Saxe. An explicating theorem prover for quantified formulas. *Draft manuscript, May*, 2004.

[FKB17] Grigory Fedyukovich, Samuel J Kaufman, and Rastislav Bodík. Sampling invariants from frequency distributions. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 100–107. IEEE, 2017.

[Flo67] RW Floyd. Assigning meaning to programs. In *Mathematical Aspects of Computer Science. Proceedings of Symposia in Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — where programs meet provers. In *European Symposium on Programming (ESOP)*, pages 125–128. Springer, 2013.

[FPMG19] Grigory Fedyukovich, Sumanth Prabhu, Kumar Madhukar, and Aarti Gupta. Quantified invariants via syntax-guided synthesis. In *International Conference on Computer-Aided Verification (CAV)*, pages 259–277. Springer, 2019.

[GBT07] Yeting Ge, Clark Barrett, and Cesare Tinelli. Solving quantified verification conditions using satisfiability modulo theories. In *International Conference on Automated Deduction (CADE)*, pages 167–182. Springer, 2007.

[GGB+22] Pamina Georgiou, Bernhard Gleiss, Ahmed Bhayat, Michael Rawson, Laura Kovács, and Giles Reger. The RAPID software verification framework. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 255–260. IEEE, 2022.

[GGK20a] Pamina Georgiou, Bernhard Gleiss, and Laura Kovács. Trace logic for inductive loop reasoning. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 255–263. IEEE, 2020.

[GGK20b] Pamina Georgiou, Bernhard Gleiss, and Laura Kovács. Trace logic for inductive loop reasoning. *Extended Version.* arXiv:2008.01387, 2020.

[GHK23] Pamina Georgiou, Márton Hajdu, and Laura Kovács. Sorting without sorts. EasyChair Preprint no. 10632, 2023.

[GKKN15] Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. The SeaHorn verification framework. In *International Conference on Computer-Aided Verification (CAV)*, pages 343–361. Springer, 2015.

[GKR18] Bernhard Gleiss, Laura Kovács, and Simon Robillard. Loop analysis by quantification over iterations. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 381–399. Springer, 2018.

[GKR20] Bernhard Gleiss, Laura Kovács, and Jakob Rath. Subsumption demodulation in first-order theorem proving. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 297–315. Springer, 2020.

[GMT08] Sumit Gulwani, Bill McCloskey, and Ashish Tiwari. Lifting Abstract Interpreters to Quantified Logical Domains. In *Symposium on Principles of Programming Languages (POPL)*, pages 235–246. ACM, 2008.

[GNT10] Michael Grottke, Allen P Nikora, and Kishor S Trivedi. An empirical investigation of fault types in space mission system software. In *2010 IEEE/IFIP International Conference on Dependable Systems & Networks (DSN)*, pages 447–456. IEEE, 2010.

[GS20] Bernhard Gleiss and Martin Suda. Layered clause selection for theory reasoning. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 297–315. Springer, 2020.

[GSM16] Arie Gurfinkel, Sharon Shoham, and Yuri Meshman. SMT-based verification of parameterized systems. In *ACM Symposium on Foundations of Software Engineering (FSE)*, pages 338–348. ACM, 2016.

[GSV18] Arie Gurfinkel, Sharon Shoham, and Yakir Vizel. Quantifiers on demand. In *Automated Technology for Verification and Analysis (ATVA)*, pages 248–266. Springer, 2018.

[HB12] Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *Theory and Applications of Satisfiability Testing (SAT)*, pages 157–171. Springer, 2012.

[HH19] Reiner Hähnle and Marieke Huisman. Deductive software verification: from pen-and-paper proofs to industrial tools. *Computing and Software Science: State of the Art and Perspectives*, pages 345–373, 2019.

[HHK+20] Márton Hajdu, Petra Hozzová, Laura Kovács, Johannes Schoisswohl, and Andrei Voronkov. Induction with generalization in superposition reasoning. In *International Conference on Intelligent Computer Mathematics (CICM)*, pages 123–137. Springer, 2020.

[HHK+22] Márton Hajdu, Petra Hozzová, Laura Kovács, Giles Reger, and Andrei Voronkov. Getting saturated with induction. In *Principles of Systems Design: Essays Dedicated to Thomas A. Henzinger on the Occasion of His 60th Birthday*, pages 306–322. Springer, 2022.

[HKV11] Kryštof Hoder, Laura Kovács, and Andrei Voronkov. Invariant generation in Vampire. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 60–64. Springer, 2011.

[HKV21] Petra Hozzová, Laura Kovács, and Andrei Voronkov. Integer induction in saturation. In *International Conference on Automated Deduction (CADE)*, pages 361–377. Springer, 2021.

[Hoa62] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[Hoa69] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[ISIRS20] Oren Ish-Shalom, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Putting the squeeze on array programs: loop verification via inductive rank reduction. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 112–135. Springer, 2020.

[JM07] Ranjit Jhala and Kenneth L. McMillan. Array abstractions from proofs. In *International Conference on Computer-Aided Verification (CAV)*, pages 193–206. Springer, 2007.

[JZ17] Dongchen Jiang and Miao Zhou. A comparative study of insertion sorting algorithm verification. In *IEEE Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)*, pages 321–325. IEEE, 2017.

[Kan18] Michael Kan. Ticketfly goes down after hacker steals customer info. *PC Mag*, 2018.

[KB83] Donald E Knuth and Peter B Bendix. Simple word problems in universal algebras. *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, pages 342–376, 1983.

[KBGM15] Anvesh Komuravelli, Nikolaj Bjorner, Arie Gurfinkel, and Kenneth L McMillan. Compositional verification of procedural programs using horn clauses over integers and arrays. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 89–96. IEEE, 2015.

[KBI+17] Aleksandr Karbyshev, Nikolaj Bjørner, Shachar Itzhaky, Noam Rinetzky, and Sharon Shoham. Property-directed inference of universal invariants or proving their absence. *Journal of the ACM (JACM)*, 64(1):1–33, 2017.

[KCSG20] Hari Govind Vediramana Krishnan, YuTing Chen, Sharon Shoham, and Arie Gurfinkel. Global guidance for local generalization in model checking. In *International Conference on Computer-Aided Verification (CAV)*, pages 101–125. Springer, 2020.

[KFG20] Naoki Kobayashi, Grigory Fedyukovich, and Aarti Gupta. Fold/unfold transformations for fixpoint logic. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 195–214. Springer, 2020.

[KG99] Christoph Kern and Mark R Greenstreet. Formal verification in hardware design: a survey. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 4(2):123–193, 1999.

[KKV15] Evgenii Kotelnikov, Laura Kovács, and Andrei Voronkov. A first class Boolean sort in first-order theorem proving and TPTP. In *International Conference on Intelligent Computer Mathematics (CICM)*, pages 71–86. Springer, 2015.

[KM97] Matt Kaufmann and J. Strother Moore. An industrial strength theorem prover for a logic based on Common Lisp. *IEEE Transactions on Software Engineering*, 23(4):203–213, 1997.

[KM16] E. G. Karpenkov and D. Monniaux. Formula slicing: Inductive invariants from preconditions. In *International Haifa Verification Conference (HVC)*, pages 169–185. Springer, 2016.

[KMM00] Matt Kaufmann, J. Strother Moore, and Panagiotis Manolios. *Computer-Aided Reasoning: An Approach*, volume 1 of *Advances in Formal Methods (ADFM)*. Springer, 2000.

[KPIA20] Jason R Koenig, Oded Padon, Neil Immerman, and Alex Aiken. First-order quantified separators. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 703–717. ACM, 2020.

[Kra22] Herb Krasner. The cost of poor software quality in the us: A 2022 report. *Consortium for Information and Software Quality (CISQ)*, pages 1–61, 2022.

[KRV17] Laura Kovács, Simon Robillard, and Andrei Voronkov. Coming to Terms with Quantified Reasoning. In *Symposium on Principles of Programming Languages (POPL)*, pages 260–270. ACM, 2017.

[KV09] Laura Kovács and Andrei Voronkov. Finding loop invariants for programs over arrays using a theorem prover. In *International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 470–485. Springer, 2009.

[KV13] Laura Kovács and Andrei Voronkov. First-Order Theorem Proving and Vampire. In *International Conference on Computer-Aided Verification (CAV)*, pages 1–35. Springer, 2013.

[Lam20] Peter Lammich. Efficient verified implementation of introsort and pdqsort. In *International Joint Conference on Automated Reasoning (IJCAR)*, pages 307–323. Springer, 2020.

[Lei10] K Rustan M Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 348–370. Springer, 2010.

[Lév14] Jean-Jacques Lévy. Simple proofs of simple programs in Why3. *Essays for the Luca Cardelli Fest*, page 177, 2014.

[LM96] Cosimo Laneve and Ugo Montanari. Axiomatizing permutation equivalence. *Mathematical Structures in Computer Science*, 6(3):219–249, 1996.

[LRCR13] Daniel Larraz, Enric Rodríguez-Carbonell, and Albert Rubio. SMT-based array invariant generation. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 169–188. Springer, 2013.

[MBTS04] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The Art of Software Testing*, volume 2. Wiley Online Library, 2004.

[MH69] John McCarthy and Patrick J Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.

[MPMW20] Anders Miltner, Saswat Padhi, Todd Millstein, and David Walker. Data-driven inference of representation invariants. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–15. ACM, 2020.

[MTK20] Yusuke Matsushita, Takeshi Tsukada, and Naoki Kobayashi. RustHorn: CHC-based verification for Rust programs. In *European Symposium on Programming (ESOP)*, pages 484–514. Springer, 2020.

[NBE+21] Tobias Nipkow, Jasmin Blanchette, Manuel Eberl, Alejandro Gómez-Londoño, Peter Lammich, Christian Sternagel, Simon Wimmer, and Bohua Zhan. Functional Algorithms, Verified, 2021.

[Nel80] Charles Gregory Nelson. *Techniques for program verification*. Stanford University, 1980.

[NO79] Greg Nelson and Derek C Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1(2):245–257, 1979.

[NR01] R. Nieuwenhuis and A. Rubio. Paramodulation-based theorem proving. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 7, pages 371–443. Elsevier Science, 2001.

[NWP02] Tobias Nipkow, Markus Wenzel, and Lawrence C Paulson. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2002.

[O'K20] Sean O'Kane. Boeing finds another software problem on the 737 max. *The Verge*, 2020.

[PMP+14] Pieter Philippaerts, Jan Tobias Mühlberg, Willem Penninckx, Jan Smans, Bart Jacobs, and Frank Piessens. Software verification with VeriFast: Industrial case studies. *Science of Computer Programming*, 82:77–97, 2014.

[PSM16] Saswat Padhi, Rahul Sharma, and Todd Millstein. Data-driven precondition inference with learned features. *ACM SIGPLAN Notices*, 51(6):42–56, 2016.

[RBSV16] Giles Reger, Nikolaj Bjorner, Martin Suda, and Andrei Voronkov. AVATAR modulo theories. In *2nd Global Conference on Artificial Intelligence (GCAI)*, pages 39–52, 2016.

[Ric53] Henry Gordon Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[RL18] Pritom Rajkhowa and Fangzhen Lin. Extending VIAP to handle array programs. In *International Conference on Verified Software. Theories, Tools, and Experiments (VSTTE)*, pages 38–49. Springer, 2018.

[Rot22] Emma Roth. Meta fined $276 million over facebook data leak involving more than 533 million users. *The Verge*, 2022.

[RS17] Giles Reger and Martin Suda. Set of support for theory reasoning. In *International Workshop on the Implementation of Logics (IWIL)*, pages 124–134. EasyChair, 2017.

[RSV21] Giles Reger, Johannes Schoisswohl, and Andrei Voronkov. Making theory reasoning simpler. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 164–180. Springer, 2021.

[RTDM14] Andrew Reynolds, Cesare Tinelli, and Leonardo De Moura. Finding conflicting instances of quantified formulas in SMT. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*, pages 195–202. IEEE, 2014.

[RV19] Giles Reger and Andrei Voronkov. Induction in saturation-based proof search. In *International Conference on Automated Deduction (CADE)*, pages 477–494. Springer, 2019.

[SG09] S. Srivastava and S. Gulwani. Program Verification using Templates over Predicate Abstraction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 223–234. ACM, 2009.

[SH20] Mohsen Safari and Marieke Huisman. A generic approach to the verification of the permutation property of sequential and parallel swap-based sorting algorithms. In *International Conference on Integrated Formal Methods (IFM)*, pages 257–275. Springer, 2020.

[She20] Michael Sheetz. Boeing takes $410 million charge to redo failed astronaut flight test if nasa requires. *CNBC*, 2020.

[Sko20] Clea Skopeliti. Thousands stranded at heathrow due to check-in systems meltdown. *The Guardian*, 2020.

[Tur36] Alan M Turing. On computable numbers, with an application to the Entscheidungsproblem. *Journal of Mathematics*, 58(345-363):5, 1936.

[Vor14] Andrei Voronkov. Avatar: The architecture for first-order theorem provers. In *International Conference on Computer-Aided Verification (CAV)*, pages 696–710. Springer, 2014.

[WL99] Andreas Wolf and Reinhold Letz. Strategy parallelism in automated theorem proving. *International Journal of Pattern Recognition and Artificial Intelligence*, 13(02):219–245, 1999.

[WLBF09] Jim Woodcock, Peter Gorm Larsen, Juan Bicarregui, and John Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys (CSUR)*, 41(4):1–36, 2009.

[WLL17] W Eric Wong, Xuelin Li, and Philip A Laplante. Be more familiar with our enemies and pave the way forward: A review of the roles bugs played in software failures. *Journal of Systems and Software*, 133:68–94, 2017.

[WPN08] Makarius Wenzel, Lawrence C. Paulson, and Tobias Nipkow. The Isabelle Framework. In *Theorem Proving in Higher Order Logics (TPHOLs)*, pages 33–38. Springer, 2008.

[WS03] Christoph Walther and Stephan Schweitzer. About VeriFun. In *International Conference on Automated Deduction (CADE)*, pages 322–327. Springer, 2003.

[WS04] Christoph Walther and Stephan Schweitzer. Verification in the classroom. *Journal of Automated Reasoning*, 32:35–73, 2004.

[WS20] Julia Carrie Wong and Olivia Solon. Google to shut down google+ after failing to disclose user data leak. *The Guardian*, 2020.

[YFG19] Weikun Yang, Grigory Fedyukovich, and Aarti Gupta. Lemma synthesis for automating induction over algebraic data types. In *Principles and Practice of Constraint Programming (CP)*, pages 600–617. Springer, 2019.

[YTGN22] Jianan Yao, Runzhou Tao, Ronghui Gu, and Jason Nieh. {DuoAI}: Fast, automated inference of inductive invariants for verifying distributed protocols. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 485–501. USENIX, 2022.

[ZCF23] Lucas Zavalía, Lidiia Chernigovskaia, and Grigory Fedyukovich. Solving constrained Horn clauses over algebraic data types. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 341–365. Springer, 2023.